AD-A037 639 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2

CANDIDATE LANGUAGES EVALUATION REPORT.(U)

JAN 77 B M BROSGOL, R E HARTMAN, J R NESTOR DAHC26-76-C-0006

UNCLASSIFIED IR-217 USACSC-AT-76-11 NL

1 OF 6

ADA037639

5.3.6

TECHNICAL DOCUMENTARY REPORT

U. S. ARMY COMPUTER SYSTEMS COMMAND

(18) USACSC (19) AT-76-11

(1)

(6)

CANDIDATE LANGUAGES EVALUATION REPORT.

(10) Authors: Benjamin M. Brosgol,
Robert E. Hartman,
John R. Nestor,
Martin S. Roth
Laurence M. Weissman

(11) Jan[redacted]77

(12) 534 p.

(9) Final rept. Oct 75- Jan 77,

(14) IR-217

(16) SX762725DY10

(17) 05

Prepared for

U. S. ARMY COMPUTER SYSTEMS COMMAND

FORT BELVOIR, VIRGINIA 22060

Prepared by

INTERMETRICS, INC.
701 Concord Ave.
Cambridge, Mass. 02138

Contract No. DAHC 26-76-C-0006 (15)

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

388 863

DISPOSITION INSTRUCTIONS

Destroy this report when no longer needed. Do not return it to the originator.

DISCLAIMER

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER USACSC-AT-76-11	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CANDIDATE LANGUAGES EVALUATION REPORT		5. TYPE OF REPORT & PERIOD COVERED Final Report for Period Oct. 1975 to Jan. 1977 6. PERFORMING ORG. REPORT NUMBER IR-217
7. AUTHOR(s) Benjamin M. Brosgol, Robert E. Hartman, John R. Nestor, Martin S. Roth, and Laurence M. Weissman		8. CONTRACT OR GRANT NUMBER(s) DAHC26-76-C-0006
9. PERFORMING ORGANIZATION NAME AND ADDRESS Intermetrics, Inc. 701 Concord Ave. Cambridge, Mass. 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 6.27.25A/SX762725DY10/05/001
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Computer Systems Command (CSCS-AT) Fort Belvoir, Virginia 22060		12. REPORT DATE January 1977 13. NUMBER OF PAGES 531
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming language, high-order language, embedded computer applications, evaluation of programming languages, FORTRAN, COBOL, PL/I, TACPOL, CS-4, JOVIAL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The main purpose of this report is to perform an evaluation of six High Order Languages -- TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I -- with respect to the Department of Defense Requirements for High Order Computing Programming Languages ("Tinman"). These requirements were determined in a previous report to be consistent with the needed programming language characteristics for Army tactical and MIS applications. An evaluation tool is developed to assist in the investigation of the technical merit

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 ABSTRACT (continued)

of the candidate languages, and management decision-making criteria are derived which take "non-technical" factors into account.

The fundamental result of this study is that, among the six HOLs considered, CS-4 comes closest to satisfying the Tinman requirements. The reason for the closeness of fit is that the language objectives receiving high priority in the Tinman (reliability, maintainability, readability) are also basic design goals of CS-4. This report recommends CS-4 as a suitable basis for a common DoD language. Because of fundamental shortcomings in each of the other five HOLs, this report recommends that these HOLs not be used as the basis for a common language.

TECHNICAL DOCUMENTARY REPORT

U. S. ARMY COMPUTER SYSTEMS COMMAND

USACSC-AT-76-11

CANDIDATE LANGUAGES EVALUATION REPORT

January 1977

Authors: B. M. Brosgol
R. E. Hartman
J. R. Nestor
M. S. Roth
L. M. Weissman

Prepared for

U. S. ARMY COMPUTER SYSTEMS COMMAND

FORT BELVOIR, VIRGINIA 22060

Prepared by

INTERMETRICS, INC.
701 Concord Ave.
Cambridge, Mass. 02138

Contract No. DAHC26-76-C-0006

DA Project SX762725DY10

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

ABSTRACT

The main purpose of this report is to perform an evaluation of six High Order Languages -- TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I -- with respect to the Department of Defense Requirements for High Order Computing Programming Languages ("Tinman"). These requirements were determined in a previous report to be consistent with the needed programming language characteristics for Army tactical and MIS applications. An evaluation tool is developed to assist in the investigation of the technical merit of the candidate languages, and management decision-making criteria are derived which take non-technical factors into account.

The fundamental result of this study is that, among the six HOLs considered, CS-4 comes closest to satisfying the Tinman requirements. The reason for the closeness of fit is that the language objectives receiving high priority in the Tinman (reliability, maintainability, readability) are also basic design goals of CS-4. This report recommends CS-4 as a suitable basis for a common DoD language. Because of fundamental shortcomings in each of the other five HOLs, this report recommends that these HOLs not be used as the basis for a common language.

FOREWORD

This document was prepared by Intermetrics, Inc., for the U. S. Army Computer Systems Command, under the authority of U. S. Army Contract No. DAHC26-76-C-0006. This document incorporates CDRL Items A004, A005, and A006 of the above-referenced contract. The DA Project Number is SX762725DY10, Task Area is 05, and Work Unit is 001. Major Benjamin D. Blood, Jr., is the Contracting Officer's Representative for the Army. Dr. Benjamin M. Brosgol is Project Manager for Intermetrics.

This study evaluates the languages TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I, with respect to the Department of Defense Requirements for High Order Computer Programming Languages ("Tinman").

The authors of this report wish to acknowledge the assistance of Judy Haigh, Jim Franklin, and Tonya Price during the preparation of the manuscript. Special thanks are due to Jim Felty for his many contributions during the editing and

production phases, and to Michelle Swanzy for her tireless "TECOing" and energetic assistance.

The material in Appendix III of this document is based on a review of an earlier version of the "Tinman" requirements, conducted by Dr. James S. Miller of Intermetrics, and sponsored by the Navy under Contract N00123-74-C-0634. In Chapter 4, paragraphs VIII.6, VIII.7, XI.1, and XI.3 are derived from work performed by Mr. Timothy A. Dreisbach under Air Force Contract F19628-76-C-0225, and paragraph VII.4 is based on a critique of CS-4 conducted by Dr. Benjamin M. Brosgol and sponsored by the above-mentioned Navy contract.

Chapters 3 through 8 of this document were produced on the Arpanet, using the TECO text editor and XOFF formatter under the TENEX operating system at USC-ISIC.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Section I. SCOPE

1. Background.

The present contract from the Army to Intermetrics calls for two general activities, occurring in sequential phases: the determination of language requirements for an Army standard programming language, and the evaluation of various high-order languages (HOLs) to determine the most suitable candidate for Tactical Data Systems applications. A Language Requirements Report [7] has been prepared in connection with the first of these activities. A major conclusion of the report is that there is no inconsistency between the basic language requirements for "general purpose" as opposed to tactical or MIS programming. As a result, the "needed characteristics" discussed in Section IV, paragraphs A through J, of the Department of Defense *Requirements for High Order* Computer Programming Languages "Tinman" (June 1976) [16], which pertain mostly to "general purpose" programming, are acceptable as the requirements for a standard Army HOL.

2. Purposes of This Report.

This report contains the results of the language evaluation phase of the contract. The work performed falls into two areas.

a. Development and Use of Evaluation Tool. An important aspect of this study is the formulation of an "evaluation tool" which can be used to assist in the process of selecting a common HOL. Chapter 2 of this report presents the tool and illustrates its use in the evaluation of six HOLs: TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I.

b. Support of DoD High Order Language Working Group. A second major purpose of this report is to support the efforts of the HOL Working Group by supplying a comparison of the above-mentioned six HOLs to the Tinman's "needed characteristics." The following subparagraphs describe the format of the comparisons presented in Chapters 3 through 8.

(1) For each language characteristic listed in the Tinman, the "degree of compliance" for each of the candidate languages is determined. The following symbols will be used to represent levels to which each language meets a requirement:

(a) T: Totally meets the requirement.

(b) P: Partially meets the requirement.

(c) F: Fails to meet the requirement.

(d) U: Unknown from the available documents if the requirement is satisfied (e.g., the requirement is strictly dependent on the operating system, libraries, or is only a management consideration).

(2) In some cases, a combination of symbols is appropriate; e.g., "PT" represents nearly total compliance, and "FU" indicates that the HOL fails to meet part of the requirement and that it is unknown from the language definition whether other parts of the requirement are met.

(3) For requirements which are "met", this report demonstrates how this is achieved (typically, the presence or absence of a particular facility), and indicates when the solution conflicts with other requirements.

(4) For requirements which are not "met" or are only partially "met", this report provides:

(a) An analysis of why the language does not fulfill or only partially fulfills the requirement.

(b) The scope of modifications that would be necessary to bring a language into compliance, and their impact on other language features.

(c) The impact of such modifications on implementation.

(5) If the requirement is not addressed in the provided language documentation, this report so indicates. In addition, features of the candidate languages are identified which are not needed to satisfy the Tinman requirements, and recommendations are presented as to whether these features should be retained or possibly eliminated.

Section II. OVERVIEW OF DOCUMENT

1. Chapter 1.

This report is divided into nine chapters and three appendices, with the current chapter providing background and introductory material.

2. Chapter 2.

Chapter 2 describes the evaluation tool and the management decision-making criteria relevant to its use (Sections I through IV) and presents an illustration of the application of the tool to the candidate HOLs (Sections V through XI). Since the Tinman's characteristics in paragraphs K, L, and M do not correspond directly to language features, the evaluation of the HOLs' "technical merit" is based just on comparisons with Tinman paragraphs A through J. In the derivation of the technical merit scores, numeric ratings are given which refine the "T", "P", "F", and "U" degrees of compliance. A rating of 0.9 or 1.0 indicates that the language (nearly) totally satisfies the requirement. Partial compliance is reflected by scores of 0.6, 0.7, or 0.8; lower scores reveal a fundamental failure to meet the requirement.

3. Chapters 3 through 8.

Chapters 3 through 8 (which do not depend on the evaluation tool) contain detailed evaluations of TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I, respectively, and recommendations concerning the suitability of modifying these HOLs to bring them into compliance with the Tinman. Each of these chapters is divided into fifteen sections.

a. Section I contains a summary of the language.

b. Sections II through XIV consist of the evaluations of the language with respect to the Tinman's paragraphs A through M. This material is organized as described in paragraph I.2b above.

c. Section XV presents a summary of the evaluation and recommendations.

4. Chapter 9.

Chapter 9 summarizes the results of this study. A synopsis of the evaluation tool, an outline of a review of the Tinman, and a set of recommendations concerning the suitability of the candidate HOLs are presented.

5. Appendix I.

Appendix I contains a sample questionnaire on HOL availability, proposed as part of the evaluation tool.

6. Appendix II.

Appendix II contains the set of Tinman's "needed characteristics" as found in [16, Section IV, paragraphs A through M].

7. Appendix III.

Appendix III provides a review of the Tinman's "needed characteristics."

CHAPTER 2

A TOOL FOR EVALUATING HIGH-ORDER LANGUAGES

Section I. APPROACH

1. Difficulties in Evaluating Languages.

Language evaluation is intrinsically a difficult and complex activity: a computer language is an intangible entity, and the factors which enter into a judgment of the merit of a HOL tend to be subjective and sometimes conflicting. The problem is that there is no such thing as a "good" language in absolute terms--a HOL which is appropriate in one environment may prove to be exactly the opposite in another. Instead, one should only speak of the merit of a language in relative terms: with respect to an application area for which the HOL is to be used, and with respect to a computing environment (hardware equipment, personnel, financial resources) in which the language is to appear.

2. Basic Approach.

The approach to language evaluation taken here attempts to cope with these difficulties by partitioning them into more manageable components (specificably addressed to application area and computing environment), and by making visible, in a quantitative fashion, the individual judgments which enter into the language evaluation process. This is not to understate the problem of accurately estimating these quantitative values; however, it is felt that the tool proposed will provide management with a handle on the basic factors which influence language selection. Moreover, if resources are available for deriving the

numeric values via an approach such as the Delphi method[1], the accuracy of the estimates will be increased. It should be emphasized, though, that we are not advocating the selection of one HOL rather than another on the sole basis of a mechanical choice of the HOL with the highest score. Rather, the evaluation tool described here is intended to serve primarily as a comprehensive guide for orderly thinking.

3. Technical and Overall Evaluations: An Introduction.

The evaluation tool proposed here is basically a methodology, or set of procedures, to be conducted in two stages. First, a technical evaluation (TE) will produce a score for the language's technical merit in meeting the requirements of a particular application area -- in this report the languages under consideration are TACPOL, CS-4, JOVIAL (J73/I), FORTRAN, COBOL, and PL/I, and the application area is that of a standard HOL for Army tactical and MIS systems. This score will then be used in the second stage in the overall evaluation (OE) of the language. The OE will take into account the computing environment in which the language will be used (viz., Army sites) via a set of management decision-making criteria.

4. Technical Evaluation.

a. Rating Matrix. The main part of the approach is the derivation of a Rating matrix, R, whose rows correspond to language goals (e.g., reliability, efficiency of object code, etc.). The value R_{ij} will be a rating of how the i^{th} feature contributes toward the j^{th} goal, and will be in the range of 0 to 10. For example, a feature like TACPOL's CODE statement, which allows a user to descend into assembly language, would have a high rating with respect to object code efficiency, but would rate poorly with respect to the goals of program reliability and transportability.

---

[1]According to the Delphi method, the members of a group of independent experts offer judgments on specific questions on two or more successive occasions. At each iteration, the amount of agreement on the chosen values is made available to the whole group, along with a list of reasons for the choices. Each group member is free to use this information in his reevaluation. This permits access by each member to the knowledge pool of the whole group, and results in a set of agreed-upon numeric values.

b. Language- and Application-Independence. It should be noted that the matrix R is both language-independent and application-independent. Language- and application-dependent vectors will be multiplied by R to obtain a score for a particular language with respect to a given application; this is described in more detail below.

c. Derivation of Rating Matrix. Three relevant issues which arise are: (1) How is the set of language features determined? (2) How is the set of language goals decided? (3) How are the ratings R_{ij} derived? With respect to language features, it was noted earlier that previous work under this contract [7] concluded that the Tinman's "needed characteristics" (Sections A through J) are suitable as the set of required features for a standard Army HOL. Thus these 78 characteristics will correspond to the rows of the matrix. The goals were determined by examining the program development process (problem specification, design, coding, debugging, maintaining) and identifying the most important objectives for a HOL at the various stages: learnability, power, efficiency, reliability, maintainability, and transportability. The entries in the Rating Matrix were derived by Intermetrics and reviewed by USACSC.

d. Language Vector. The Rating Matrix, R, say with m rows and n columns (here m=78 and n=6), is used in conjunction with two vectors for the purpose of the Technical Evaluation. The Language vector, L, is a row vector with m components (the number of language features). For each candidate language C, there is a Language vector L^C; the i^{th} element in L^C is a weighting between 0.0 and 1.0 (inclusive) which reflects how well language C satisfies the i^{th} Tinman requirement. It should be noted that the product $L^C * R$ is a row vector with n components; its j^{th} element is a score between 0 and 780 giving the overall rating of language C with respect to the j^{th} goal.

e. Application Vector. The Application vector, A, is a column vector with n components (the number of language goals). For any given application area, U, there is an application vector A^U satisfying the following conditions:

(1) $A_j^U \geq 0$ for $j = 1, \ldots, n$

(2) $\sum_{j=1}^{n} A_j^U = 1.0$

Intuitively, A_j^U is the relative importance of the j^{th} goal for the application U. For example, for tactical applications, the goal of object code efficiency might receive a higher weight than the goal of transportability. The reason for requiring the weights in A^U to sum to 1.0 is that goals frequently overlap (e.g., learnability, reliability, maintainability), and allowing non-normalized values in A^U would result in overemphasis of such goals (i.e., effectively, the same goal would be counted several times).

f. Intermediate Vector. The product $R * A^U$, denoted IV^U, is a column vector, language-independent, with m components. Its i^{th} element is a score between 0 and 10 which reflects how well the i^{th} Tinman requirement contributes toward application area U.

g. Technical Merit. The product $L^C * IV^U$ yields a scalar value which reflects the technical merit of language C with respect to application area U. The maximum value for this product is obtained when C scores 1.0 for each entry in IV^U; we denote this summation by IV_*^U; i.e., $IV_*^U = \sum_{i=1}^{m} IV_i^U$ where m is the number of features. Thus the range for $L^C * IV^U$ is 0 through IV_*^U. Since, for convenience of reference as well as for potential use in the overall evaluation we are interested in the range 0 through 100, we define the technical merit of language C, with respect to application area U, to be $100 * (L^C * IV^U)/IV_*^U = K(U) * (L^C * IV^U)$ where the transformation constant $K(U) = 100/IV_*^U$.

h. Absolute Requirements. For certain applications, a number of requirements are "absolute" in that a language lacking these requirements cannot be used. As an example, for list processing applications the following requirement (from D6 in the "Tinman") is absolute: "The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure." For tactical applications, the "Tinman" requirement J3 (Machine Language Insertions) is absolute. Thus, for the purposes of this report, any language which does not satisfy J3 would be excluded from further consideration. FORTRAN and PL/I -- as described by their reference documents as opposed to their implementations -- fall into this category.

5. Overall Evaluation.

a. Introduction. The technical merit of a given language with respect to a particular application area is derived in the Technical Evaluation stage; the Overall Evaluation takes this score into account as well as other factors specific to the computing environment in which the language is to be used. These factors interact so as to affect the cost of producing programs in the given language. The overall evaluation attempts to account for these effects and to help estimate the costs involved for any candidate HOL.

b. Breakdown of Costs. The non-hardware expenses incurred during the development of a software system can be classified into the following categories:

(1) Language acquisition. This entails obtaining a language processor, support tools and documentation so that the system can be programmed.

(2) Programmer training. Available or new personnel may have to be trained to use the language selected.

(3) System development. This area entails the following activities:

(a) Defining the system performance requirements.

(b) Designing the system.

(c) Coding the system in the given language(s).

(d) Testing, debugging, and verifying the software.

(e) Maintaining and upgrading the system, perhaps in response to changes in the performance requirements.

(f) Transporting the system perhaps to new hardware configurations.

(4) Indirect expenses. This category includes expenses resulting from not meeting the system performance requirements -- e.g., costs stemming from late or incorrect software.

c. Management Decision-Making Criteria. The actual costs associated with b(1) through b(4) above depend on a variety of factors. Some of these factors relate directly to the language chosen for system implementation; we term these factors "management decision-making criteria" since they may be used by management to help guide the language selection process. The set of criteria considered in this report are technical merit; compiler, support tool, and documentation availability; standardization status; industry support; and availability of trained programmers.

d. Outline of Overall Evaluation. The approach suggested here is to utilize a set of values for each candidate language's compliance with the management decision-making criteria. This set will be applied in order to determine the impact of these criteria upon the various costs associated with software system development. That is, for candidate language C, application area U and computing environment E, there will be a Language Management vector $LM^{C,U,E}$ whose i^{th} component is a rating of how well language C meets the i^{th} decision-making criterion; $0 \leq LM_i^{C,U,E} \leq 100$ for each i. The cost associated with the j^{th} "expense component" (language acquisition, programmer training, system development, indirect expenses) will then depend on the values in $LM^{C,U,E}$. Let $X_j^{C,S}$ be the expense associated with the j^{th} cost component for candidate language C and software system S; then $X_j^{C,S}=f_j(LM^{C,U,E},S)$ for some function f_j. If the functions f_j could be approximated, the user of the evaluation tool could obtain estimates on the effect of the language upon system cost; these estimates could then be used to assist in the process of HOL selection.

6. Limitations of the Evaluation Tool.

The evaluation tool described in this report has good potential for lending assistance during the HOL selection process, and perhaps its greatest benefit emerges from the way it induces an organized and orderly investigation into the factors which influence the choice of a language. Nevertheless, there are several limitations on the applicability of the tool; these are outlined in the following paragraphs.

a. Numeric Approach. The problems associated with obtaining quantitative estimates for the various components of the tool are severe. Techniques such as the Delphi method are helpful; however, a fair amount of research is needed in order to obtain an information baseline to guide the participants in these techniques. In addition, certain aspects of the Tinman document caused complications in the derivation of numeric scores during the Technical Evaluation stage for this report. One problem is the absence of precise definitions for the terms used in the various "needed characteristics". Although it might be argued that the provision of such definitions would overemphasize individual features at the expense of general requirements, it is nevertheless true that the ambiguities in the Tinman cause many of the scores to depend on specific interpretations. (A review of the Tinman, including illustrations of ambiguity, is given in Appendix III of this report.) Another problem in using the Tinman is that most of the "needed characteristics" are not single requirements. Typically a characteristic comprises a set of requirements, and often these constituents vary in their importance; this makes it a difficult decision to compose a single score for the HOL's compliance with the characteristic.

b. "New" Languages. In the application of the tool, the technical evaluation is based solely on the defining document for the language, while the overall evaluation takes into account the history of the language's use. There are problems, however, when the defining document introduces revisions to an existing language (in this report, FORTRAN, COBOL and PL/I illustrate this situation). The difficulty is that the language being technically evaluated may be considerably different from the language actually in use. In this report we account for such circumstances by describing the degree of difference between the languages as defined and as implemented; the more significant this difference, the lower the non-technical ratings we should use during the overall evaluation.

c. Modifiability Constraints. A basic aspect of the tool is that each candidate language is evaluated "statically" -- i.e., with respect to its closeness to the requirements but not with respect to the ease or difficulty of making modifications to bring it into compliance. In other words, the tool as described here is applicable to determining the "best fit" to the requirements, but is not directly applicable to determining how to modify the language so

that it meets the requirements. It should be emphasized that discussions concerning the scope of modificaations needed to bring the various candidate HOLs into compliance with the Tinman are supplied in the subsequent chapters of this report; however, these are not utilized in the evaluation tool. An extension of the tool to take such data into account appears to be a fruitful area for further research.

Section II. LANGUAGE-INDEPENDENT ASPECTS OF THE TECHNICAL EVALUATION

1. Introduction.

To carry out the Technical Evaluation stage of the evaluation of language C for application area U, we must derive the language vector L^C, the Rating matrix R, and the Application Vector A^U. This section presents A^U and R, which are independent of the language being evaluated; the vectors for the various candidate HOLs are given in Sections V through X.

2. Application Vector.

a. Selection of Language Goals. The application vector A^U consists of non-negative numeric weightings which sum to 1.0; the j^{th} element in A^U reflects the relative importance of the j^{th} goal in application area U. The number and variety of the goals which are relevant to high-order languages are quite large, and choosing from these goals a set to be used in the language evaluation is not a straightforward decision. In making the selection of goals, we attempted to avoid choosing ones which overlapped (however, some redundancy is inevitable due to the interrelationships among language objectives). In addition, we were guided by the use to be made for the goals -- viz., in the rating matrix and application vector. Thus, the ability to estimate how well a language feature contributed toward meeting a goal, or how appropriate a goal was in a given application area, were important in guiding the selection. The set of goals to be used in this study are: learnability, power, efficiency, reliability, maintainability, transportability. As mentioned earlier in the Chapter (paragraph I.4.c), these goals are applicable at the various stages of the program development process (problem specification, design, coding, debugging, maintaining). Learnability is relevant at all stages; power is most applicable during design and coding; efficiency and reliability are relevant in coding, debugging, and maintaining; maintainability is applicable during debugging and maintaining; transportability is relevant during program maintenance.

b. Definition of Language Goals.

(1) Learnability. The goal of learnability encompasses simplicity, uniformity of notation, and consistency of rules. A learnable language will have a short, clear defining document and will make possible the production of a small fast compiler.

(2) Power. Power implies both generality (i.e., applicability to a diversity of programming areas) and problem-oriented high-level notation. Extensibility in a HOL contributes toward power.

(3) Efficiency. The efficiency of a language measures how well the code generated by a good compiler will compare (in storage and speed) to that produced by a good assembly language programmer. With respect to the degree to which a given HOL feature contributes toward efficiency, the important consideration is the quality of the code which can be produced, and whether runtime overhead is implied.

(4) Reliability. By "reliability" we mean the ability of a language to permit and facilitate the writing of demonstrably correct programs. Compile-time checking, program test and debug features, and facilities which allow formal proofs of correctness, are some of the factors contributing toward this goal.

(5) Maintainability. Program maintainability includes the goals of readability and ease of constructing large systems. Facilities which support maintainability range from source listing formatters to language features such as data abstraction which help to localize the program changes in an evolving system.

(6) Transportability. By "transportability" we mean the ability to move a program intact from one host-target-machine environment to another and obtain the same result (except for possible differences in the precision of arithmetic values caused by differences in machine word lengths). Transportability implies an absence of "undefineds" in the language specification.

c. Weightings for the Application Vector Components. The weightings derived for the various goals, when applied to the application area of a standard Army HOL for tactical systems, are as follows:

Learnability:	.10
Power:	.15
Efficiency:	.20
Reliability:	.25
Maintainability:	.20
Transportability:	.10

3. Rating Matrix.

The Rating matrix has 78 rows (the number of "needed characteristics" in Section IV of the Tinman) and 6 columns (the number of goals). Entries in the matrix are values between 0 and 10, with the following interpretations:

Weighting value	Explanation
10	The feature is relevant to the goal and strongly helps attain the goal.
7.5	The feature is relevant to, and does not conflict with, the goal.
5	The feature has no effect on either attaining or defeating the goal.
2.5	The feature is relevant to, and conflicts with, the goal.
0	The feature is relevant to the goal and strongly helps defeat the goal.

Table I displays the Rating matrix; the three-character codes along the left side correspond to the individual "needed characteristics" in the Tinman.

4. Product of Rating Matrix and Application Vector.

a. The intermediate result $IV^U=R*A^U$ in the technical evaluation is a 78x1 column vector. The i^{th} entry is a measure of the suitability of the i^{th} Tinman requirement with respect to the application area of a standard Army HOL; it is a value between 0 and 10. The entries in this vector are displayed in Table II, and these values reflect directly the desirability of the various Tinman characteristics with respect to the given application area. Here we see that type definitions (E5) and data defining mechanisms (E6)

TABLE I. RATING MATRIX

	LEARNABILITY	POWER	EFFICIENCY	RELIABILITY	MAINTAINABILITY	TRANSPORTABILITY
A01	7.5	8.5	5.0	10.0	6.5	6.5
A02	7.5	9.5	5.5	9.5	7.5	7.5
A03	6.0	5.5	6.0	7.5	6.5	9.5
A04	3.5	8.0	4.5	4.5	4.5	8.5
A05	6.0	6.0	5.0	5.5	5.5	6.0
A06	9.0	9.5	4.0	9.5	8.5	6.5
A07	7.0	9.5	3.0	9.5	7.5	7.5
B01	8.5	8.5	6.5	8.0	7.5	7.0
B02	8.5	7.5	6.5	8.0	7.5	7.0
B03	8.5	8.5	5.0	8.0	7.5	7.0
B04	8.0	7.0	7.0	7.5	7.5	6.5
B05	6.0	5.0	3.5	8.0	6.5	5.0
B06	8.5	8.5	6.5	8.0	8.0	5.5
B07	3.0	9.0	7.5	3.0	3.5	5.5
B08	9.0	6.0	6.5	9.5	9.0	5.5
B09	7.0	6.0	4.0	7.5	7.5	5.5
B10	7.5	9.5	5.0	7.5	7.5	9.0
B11	5.5	7.5	7.0	6.0	6.0	5.5
C01	7.5	5.5	8.0	7.5	7.5	7.5
C02	8.5	4.0	5.0	8.5	8.5	5.0
C03	8.5	8.5	4.5	8.5	8.5	5.0
C04	6.5	8.5	5.5	7.5	7.5	5.5
C05	8.0	7.5	5.0	8.0	8.0	5.5
C06	9.0	7.5	4.5	9.0	9.0	5.5
C07	8.5	9.0	6.0	8.5	8.5	5.5
C08	7.0	8.5	5.5	7.0	7.0	5.5
C09	7.0	8.5	5.5	7.0	7.0	5.5
D01	8.0	8.5	5.5	8.0	8.0	5.5
D02	8.5	8.0	5.5	8.5	8.5	7.5
D03	7.0	9.0	5.5	7.5	7.5	5.5
D04	8.0	8.0	4.5	8.5	8.5	5.5
D05	9.0	8.5	5.0	8.5	8.5	6.0
D06	4.5	5.5	9.5	4.5	4.5	6.0
E01	6.5	10.0	6.0	7.5	8.0	6.0
E02	9.0	7.5	6.0	9.0	9.0	6.5
E03	9.0	5.0	5.0	9.0	9.0	5.0
E04	5.5	8.0	5.0	5.5	5.5	5.0
E05	9.0	9.5	5.5	9.5	9.5	6.5
E06	9.0	9.5	5.5	9.5	9.5	6.5
E07	9.0	5.0	5.0	9.0	9.0	5.0
E08	9.0	9.5	5.5	9.0	9.0	5.5

TABLE I. RATING MATRIX (CONTINUED)

	LEARNABILITY	POWER	EFFICIENCY	RELIABILITY	MAINTAINABILITY	TRANSPORTABILITY
F01	7.5	9.0	5.0	7.5	7.5	5.0
F02	9.0	8.5	5.0	9.5	9.5	5.5
F03	8.5	7.0	7.5	8.5	8.5	5.0
F04	7.5	8.5	6.5	9.0	9.5	6.0
F05	7.5	9.0	5.5	8.5	9.5	5.5
F06	7.5	9.0	5.0	8.5	9.5	5.5
F07	9.0	6.0	4.0	9.0	9.0	9.5
G01	7.0	9.0	4.5	9.0	9.0	8.0
G02	7.5	8.0	8.0	6.5	6.5	5.0
G03	8.5	8.0	7.5	8.5	8.5	5.0
G04	8.5	8.0	7.5	8.5	8.5	5.0
G05	6.0	9.5	4.5	6.0	6.0	5.0
G06	7.5	9.5	3.5	7.5	7.5	8.5
G07	7.5	9.5	4.5	7.5	7.5	8.5
G08	7.5	9.5	4.5	7.5	7.5	8.5
H01	9.0	5.0	5.0	9.0	9.0	5.0
H02	9.0	4.0	5.0	9.0	9.0	5.0
H03	5.5	5.0	5.0	5.5	5.5	9.0
H04	7.5	5.0	5.0	7.5	7.5	5.0
H05	7.5	5.5	5.0	7.5	7.5	5.0
H06	8.0	5.0	5.0	8.0	8.0	5.0
H07	6.0	5.0	5.0	6.0	6.0	5.0
H08	8.0	5.0	5.0	8.0	8.0	5.0
H09	7.5	5.0	5.0	7.5	7.5	5.0
H10	8.0	5.0	5.0	8.0	8.0	5.0
I01	8.0	5.0	5.0	8.0	8.0	7.5
I02	7.5	5.0	5.5	7.5	7.5	6.5
I03	6.5	7.5	6.0	6.5	6.5	6.0
I04	6.5	8.0	7.5	6.5	6.5	7.0
I05	8.0	7.5	4.5	8.0	8.0	6.5
I06	5.0	5.0	5.0	5.0	5.0	8.5
I07	5.5	5.0	5.0	5.0	5.0	4.5
J01	5.0	4.5	9.5	5.0	5.0	5.0
J02	7.5	5.0	8.0	7.5	7.5	5.0
J03	3.5	9.5	10.0	.5	.5	.0
J04	5.0	5.0	9.5	5.0	5.0	4.5
J05	5.0	5.0	9.0	5.0	5.0	5.0

score most highly (about 8.4), while the machine language insertions requirement (J3) emerges as least desirable (4.0). This latter conclusion may seem surprising, in light of the high dependence currently on direct code in some tactical system software. However, it reveals that the weightings in the application vector (paragraph 2c above) represent compromises between a variety of intended applications, since the area in question (that of a standard language for tactical software) is fairly broad. In one of the particular subareas concerned with tactical applications, the weightings for power and efficiency would be somewhat higher, and those for other goals appropriately lower, than the actual values in the Application Vector. For such a subarea, machine language insertions would receive a higher score in the Intermediate Vector.

b. The sum of the entries in IV^U is 541.4; thus the transformation constant (see paragraph I.4.g in this chapter) is K(U) = .185.

TABLE II. PRODUCT OF RATING MATRIX AND APPLICATION VECTOR

	1	2	3	4	5	6	7	8	9	10	11
A	7.47	7.90	6.75	5.32	5.57	7.85	7.35				
B	7.62	7.47	7.32	7.27	5.85	7.57	5.15	7.82	6.32	7.45	6.32
C	7.30	6.77	7.35	6.95	7.07	7.52	7.77	6.77	6.77		
D	7.32	7.72	7.07	7.27	7.60	5.80					
E	7.42	7.92	7.20	5.72	8.35	8.35	7.20	8.02			
F	6.97	8.00	7.72	8.07	7.77	7.67	7.60				
G	7.80	6.97	7.87	7.87	6.12	7.10	7.30	7.30			
H	7.20	7.05	5.67	6.38	6.45	6.65	5.55	6.65	6.38	6.65	
I	6.90	6.62	6.50	6.97	7.07	5.35	5.00				
J	5.82	6.97	4.00	5.85	5.80						

Section III. LANGUAGE-INDEPENDENT ASPECTS OF THE OVERALL EVALUATION

1. Background.

a. High Cost of Software. The basic issue which underlies this study is that of the high costs of software, and, in particular, the role of the language with respect to these costs. There are two factors involved with high costs which must be considered. First, part of the expense of software can be attributed to the expansion in the applications for which computers are being used. Second, part of the costs are indirect, due to inefficiencies in the production of the software which are avoidable with the proper managerial/technical facilities. The choice of proper language is relevant to both factors. With respect to the first area, the language must contain features that enable the necessary applications to be programmed at all. Concerning the second, in view of the fact that program verification and maintenance have been identified as the software development stages demanding the most resources -- particularly, programmer time -- the language (or the tools supporting its use) must promote the use of sound programming methodology.

b. "Non-Technical" Factors in Choosing a Language. The evaluation tool used in this report accounts for these factors associated with high software costs, primarily through the technical evaluation stage. Clearly, however, there are other criteria which enter into a language selection decision, having to do mainly with the existence of the language, and the history of its use (e.g., compiler and support tool availability). Unfortunately, these other criteria do not always correlate directly with the technical merit of the language. In fact, the older the language, the more likely it is for the language to have wide usage and good support tools, but the less likely for it to fare well against technical standards oriented around state-of-the-art research.

c. The Importance of a HOL's Technical Merit. The decision as to whether the benefits of an existing language outweigh the advantages of a technically superior but unimplemented language[1] must be based on the individual

[1] In the term "unimplemented language", we include both a language which has no implementation, and a language which has an existing implementation, but which has been modified in its design for technical improvement and for which these modifications have not been implemented.

circumstances surrounding the language evaluator's computing environment. For a private industry which is constrained by a particular set of computing equipment, the costs associated with implementing a technicably superior but unimplemented language would probably outweigh the benefits, and the use of an existing language might be more cost-effective. However, when one considers software procurement and development agencies such as the Army, or DoD, then technical merit tends to be the decisive criterion, based on the relatively small costs involved with language implementation compared with expected savings from the use of a better language. With DoD software costs in the range of $3 billion annually, the potential savings for each percent come to $30 million per year. Just a one-percent savings from the use of a technically superior language would more than offset any needed development and implementation costs.

d. Approach to Defining the Management Decision-Making Criteria. For the purposes of the Overall Evaluation stage of the evaluation tool, a set of management decision-making criteria is described in the next paragraph. These criteria take into account technical factors as well as considerations relating to existing implementations, including the relevant requirements from the "Tinman's Needed Characteristics," Sections K, L, and M. The discussions of the various criteria define procedures which may be carried out to establish quantitative scores for any candidate HOL's Language Management vector. However, because the effort required to perform these procedures for the non-technical factors was beyond the scope of the contract, this report does not present numeric scores for the candidate HOLs. Instead, qualitative assessments are given.

2. Management Decision-Making Criteria.

Paragraphs a through e below describe procedures for obtaining quantitative estimates, in the range 0 to 100, for any candidate HOL's degree of compliance with the management decision-making criteria.

a. Technical Merit. The technical merit of the language is the result of the technical evaluation stage. It is the only criterion which is independent of the existence of an implementation for the language.

b. Availability. The score for a language's availability is derived from the results of a questionnaire (Appendix I) distributed within the Army. It is intended that copies of this questionnaire be sent to each programming group

manager conducting an in-house project, and to each technical monitor who oversees a contractor programming effort.

(1) The questionnaire consists of two parts, and a separate questionnaire is to be filled out for each language in use by the members of the group. The first part requests general information on the reasons for using the language, and the kinds of hardware on which the language is implemented. This information, though not directly impacting the language's score for the availability criterion, could be used subsequently in an analysis of the reasons for the presence of those languages which are available. The first part of the questionnaire also requests that the respondent specify the total number of programmers using the language. This number will then be used as a <u>weighting factor</u> in the determination of the availability score. The way this is done is as follows. Assume that there are r respondents and that a total of c candidate languages are mentioned in the responses. We denote by P(i,j) the number of programmers in the site of the i^{th} respondent who use the j^{th} language. (The questionnaire explains how to avoid duplicating a count when the same programmer uses several languages parttime.) P(*,j) is the total number of programmers using language j -- i.e., $P(*,j)=\sum_{i=1}^{r} P(i,j)$.

The <u>weighting factor</u> W(j) for the j^{th} language is defined to be P(*,j) / P(*,jmax) where jmax is the index of the language with the largest total number of programmers: i.e., $P(*,j) \leq P(*,jmax)$ for j=1,2,...,c. The use of this weighting factor is explained below.

(2) The second part of the questionnaire requests scores and weightings relevant to the availability of the language. This part comprises four sections: compiler, tools, documentation, and other. The respondent, by supplying the information requested, will induce separate scores (between 0 and 10) for the availability of the compilers, tools, documentation, and any other values he wishes to include. He is asked to provide weightings (values between 0.0 and 1.0 which sum to 1.0) for these four components

which reflect the relative importance of these components for his particular programming group. The result of summing the weighted scores is a score for the availability of language j at site i; we denote this score by S(i,j).

(3) The score for the availability of the j^{th} language, over all sites, is then calculated using the derived weighting factors W(j) and availability scores S(i,j): this result is

$$W(j) * \frac{10}{r} * \sum_{i=1}^{r} S(i,j),$$

and will be a value between 0 and 100. Note that, for the most available language (the $jmax^{th}$), the weighting factor is simply 1.

(4) It should be realized that one of the most critical aspects of this process, in terms of the impact on the final score for availability, is how to weight the responses. (The weighting question arises also in the derivation of a single score for each respondent, but we solved this problem by asking the respondent himself to supply the weightings for the various constituents according to the requirements and priorities of his project.) For example, one approach is to consider all respondents equal -- but this fails to distinguish availability for large vs. small projects. Our method here is to account for this by using as the weighting factor the percentage of programmers using the language. But even here there are several possible techniques. For example, the weighting could be a pure percentage -- if 50% use language A, 30% use language B, and 20% use language C, the weights would be .5, .3, and .2, respectively. This has the effect of partitioning the maximum availability score into components: i.e., language A could then score no more than 50, B could do no better than 30, and C is constrained to be less than or equal to 20. This was not the approach used with the score for technical merit, however; in the latter, each language was judged independently against a single model. We have attempted to use a similar strategy here; the single model in this case is the language with

maximum availability. Thus the weighting factor for this language is automatically 1, and the others are derived relative to this standard. In the above example, the weights for languages A, B, and C would be 1 (.5/.5), .6(.3/.5), and .4(.2/.5).

c. Standardization Status. One of the objectives of adopting a common HOL is to eliminate or prevent incompatible dialects. To derive a score for this criterion, we use the following factors:

(1) The existence of a complete, unambiguous definition, with a minimum of implementation dependencies, which specifies not only the legal programs in the languages but the actions to be taken when a program is illegal.

(2) The existence of a formalized procedure (e.g., a set of test programs -- both legal and illegal) to be used for validating a compiler vs. the language definition.

(3) The existence of a well-defined, orderly procedure for permitting the evolution of the language based on user feedback and advances in the state-of-the-art in language design and programming methodology.

(4) The existence of a committee, with DoD involvement, embodied with responsibilities for maintaining the language definition standard and answering questions from implementors.

(5) The existence of a journal (e.g., COBOL's Journal of Development) devoted specifically to the language.

Each factor will be rated, from 0 to 10, based on the standardization status of the language being evaluated. These scores will be weighted as follows: .4 for (1), .15 for (2), .15 for (3), .2 for (4), and .1 for (5). The following weighted sum (between 0 and 100) will be used as the value for standardization status:

$$10 * \sum_{i=1}^{5} \text{Weight}(i) * \text{Score}(i)$$

d. Industry Support.

(1) This is a difficult factor to quantify. Ideally, a questionnaire of the sort used to determine availability (Appendix I) would be useful, but it is extremely doubtful that a company would voluntarily offer such information, due to the possibly privileged nature of the data, and to the time required to complete the questionnaire. The alternative suggested here is to attempt to obtain information concerning at least the production of compilers for the various languages under consideration. This can be done by sending a brief questionnaire to the 100 largest companies in the data processing industry. This questionnaire will request that the company (i) list each compiler and language-specific software aid currently available, and (ii) specify the hardware environment needed for each item in (i).

(2) The reduction of the data obtained from responses to these questionnaires, to derive a single score between 0 and 100, proceeds as follows. First, to give compilers more weight than support tools, we assign three points to each compiler and one point to each support tool. For any given response, such a point count is multiplied by the number of different machines for which the compiler or tool is implemented. Let P(i,j) be the total number of points derived from the i^{th} respondent with respect to the j^{th} language;

$$P(*,j)=\sum_{i=1}^{100} P(i,j)$$

is the total number of points for the j^{th} language.

(3) The basic idea is to measure industry support relative to some standard. As in the calculation of availability, we here use as the standard the language most supported (as revealed in the responses). This is done in two ways (one which treats all companies as equal, the other which weighs larger companies more heavily); the average of these two is then used as the score for industry support. According to the first method, we define jmax as the index of the language with

the highest total points; i.e., $P(*,jmax) \geq P(*,j)$ for all j. Then the score for industry support for language j, denoted $IS_E(j)$ -- The E subscript denotes the fact that all companies are considered equal -- is simply $100*P(*,j)/P(*,jmax)$. The second method determines the score for industry support of language j, taking weighting into account; this score is denoted $IS_W(j)$. To calculate the weights, assume that the total data processing revenues for the i^{th} company are D(i) dollars (this information is obtainable from various sources, such as [11]). Let

$$D(*) = \sum_{i=1}^{100} D(i)$$

and define the weight W(i) to be D(i)/D(*). Let the weighted point total for company i and language j, denoted WP(i,j), be the product W(i)*P(i,j). Define WP(*,j) to be

$\sum_{i=1}^{100} WP(i,j)$, the weighted point total for language j. Analogous to the first method, let jmaxw be the index of the language with the largest weighted point total - i.e., $WP(*,jmaxw) \geq WP(*,j)$ for all j. Finally, the score $IS_W(j)$ is defined to be $100*WP(*,j)/WP(*,jmaxw)$. And the score for the industry support of language j, or IS(j), is simply $(IS_E(j) + IS_W(j))/2$.

(4) As an example of this calculation, assume that Company 1's response results in 5 points for Language A, 3 points for Language B, and 0 points for Language C; for Company 2, we have 1 point for A, 2 for B, 5 for C; for Company 3, we have 2 points for A, 5 for B, and 1 for C. Thus, P(*,A) = 8, P(*,B) = 10, and P(*,C) = 6; jmax = B; and $IS_E(A) = 100*8/10$, $IS_E(B) = 100*10/10$, and $IS_E(C) = 100*6/10$. Assume that Company 1's percentage of total revenue is 50%; for Company 2, it is 30%; and for Company 3, it is 20%. The weighted point totals are as follows: for Company 1, A has 2.5 points and B has 1.5; for Company 2, A has 0.3 points, B has 0.6, and C has

1.5; for Company 3, A has 0.4 points, B has 1.0, and C has 0.2 points. Now, WP(*,A) = 3.2, WP(*,B) = 3.1, and WP(*,C) = 1.7; jmaxw = A; and $IS_W(A) = 100*3.2/3.2$, $IS_W(B) = 100*3.1/3.2$, $IS_W(C) = 100*1.7/3.2$. Thus, IS(A) = 90, IS(B) = 98, and IS(C) = 57.

e. Availability of Trained Programmers. A score for the availability of programmers for a language can be obtained using questionnaires distributed to programming group managers within the Army. Each respondent would be requested to supply the following information for each language used in his project:

(1) How many programmers are currently available to use the language on current (or planned) activities?

(2) How many programmers for the language are required to perform the current (or planned) activities?

Let PA(i,j) be the numbers of Programmers Available at the i^{th} site for the j^{th} language, and let PR(i,j) be the number of Programmers Required. Define the total number of programmers available for the j^{th} language, PA(*,j), as $\sum_i PA(i,j)$; similarly, $PR(*,j) = \sum_i PR(i,j)$. The score for the availability of the j^{th} language is now defined as 100 if $PA(*,j) \geq PR(*,j)$, or 100*PA(*,j)/PR(*,j), otherwise.

3. Effects of Management Decision-Making Criteria on Cost.

a. Scope. As described in the outline of the Overall Evaluation (paragraph I.5d of this chapter), we intend for the scores in the Language Management vector to be used in the projection of estimates for the various cost categories. A fundamental step in the process is the derivation of the functions that relate the Language Management elements to the costs. However, the effort required to carry out this derivation was beyond the scope of the contract. As a result, the relationships between the management decision-making criteria and the software cost categories are described below in a qualitative manner.

b. Language Acquisition. The costs associated with acquiring a language processor, support tools, and documentation, are obviously strongly dependent on the availability of these elements. If they are not available in the computing environment in which they will be used, then the costs are influenced directly by industry support and standardization status, since it may be possible to obtain the language commercially. The technical merit affects acquisition costs if the compiler and tools must be implemented "from scratch" or transported from a different computing environment. In the latter case, the "maintainability" and "transportability" components of the technical merit will be most directly relevant. Programmer availability has minor impact on acquisition costs (programmer availability pertains to the language being acquired).

c. Programmer Training. Programmer availability is the factor that most affects training costs. Training presupposes the existence of a language processor, support tools, and documentation; the costs for acquiring these elements were taken into account in paragraph b above. Standardization status and industry support affect programmer training costs only indirectly, through their impact on language acquisition. A language's technical merit can have a substantial influence on programmer training costs; the "learnability" and "power" components of the technical merit are the scores most directly relevant.

d. System Development. The implementation of a software system requires a language processor environment and trained programmers; the factors affecting these costs were considered in paragraphs b and c above. Language and programmer availability, standardization status, and industry support influence system development costs indirectly, through their influence on language acquisition and programmer training. The major factor that directly affects system development costs is the language's technical merit; as described in paragraph II.2a of this chapter, each component of the technical merit is applicable at some stage of the system development process. Of special relevance are the "reliability" and "maintainability" components of technical merit, in view of the relatively high percentage of software life-cycle costs that are devoted to debugging and maintenance.

e. Indirect Expenses. The indirect expenses (i.e., the costs resulting from software that does not meet its performance requirements) are by their nature the most difficult to estimate. Relationships between the management decision-making criteria and these expenses are system-specific; in fact, these expenses are influenced more by the way in which the system development is managed than by the management decision-making criteria. As a result, a study of the subject of indirect expenses proved to be outside the scope of this report.

4. Directions for Further Work.

There are two aspects of the Overall Evaluation that require further study before the techniques proposed become fully practicable. First is the derivation of quantitative measures for the various management decision-making criteria; this could rely on paragraph II.2 as its basis. Second, and more difficult, is the establishment of functions that relate the scores for the decision-making criteria to the actual costs involved with system implementation. A possible approach here is to use the actual costs associated with the implementation of *existing software systems*, so that a relationship between the decision-making criteria and these costs could be fitted.

Section IV. SUMMARY OF EVALUATION TOOL

1. Introduction.

a. This section is intended as a guide to using the evaluation tool. Although, for ease of exposition, the description takes the form of a single sequence of steps, we expect that, in practice, the application of the tool will be an iterative process. That is, any quantitative results obtained should be checked vs. the evaluator's intuitive judgments concerning the language, application area, and computing environment. If differences occur, the evaluator can and should trace the causes in the tool itself. Possible reconciliations for such differences are to change the sets of goals, features, or management decision-making criteria; to revise scores or weightings in the various vectors or the rating matrix; or to revise one's intuitive judgment after re-examination of the evidence. It should be emphasized that making modifications to the components of the tool in such a fashion is consistent with the main purposes of the tool - to <u>assist</u> (not to replace) the evaluator in formulating judgments on a language and to increase his understanding of the reasons behind those judgments.

b. The main steps in applying the evaluation tool are summarized in Figure 1. The ovals depict qualitative decisions, and the rectangles stand for quantitative ones. The arrows between components represent dependence; "$\boxed{A} \rightarrow \boxed{B}$" means that A must be carried out before B. Since the determinations of goals, features, and management criteria are logically independent, there is considerable flexibility in sequencing the steps of an evaluation.

c. There is a fairly wide variation in the kinds of activities implicit in the several stages of the evaluation tool's application. The component of Figure 1 marked "Compute Technical Merit" can be carried out mechanically, given the necessary matrix and vectors. A summary of these steps is given in Table III. On the other hand, the determination of goals, features, and criteria, and the derivation of scores and weightings for the rating matrix and various vectors, are not mechanizable, and a technique such as the Delphi method (supplemented by a questionnaire analysis as described in paragraph 2.III.2) may be appropriate if resources are available. (This last qualification is relevant, since the number of goals, features, and criteria which are needed to ensure completeness may result in extremely large vectors or matrices.

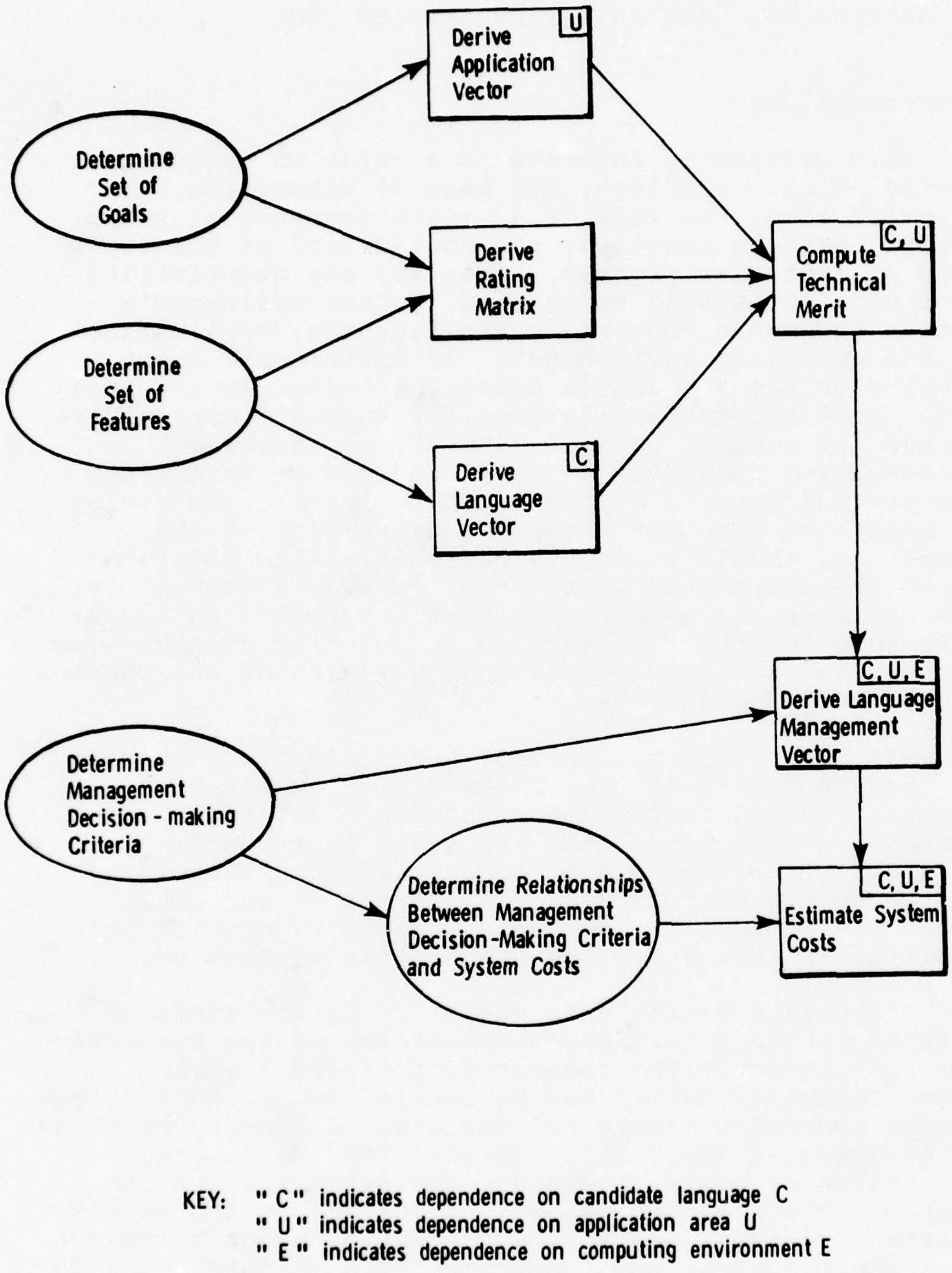

Figure 1. Stages in the Application of the Evaluation Tool

TABLE III. TECHNICAL EVALUATION

	R *	A^U →	IV^U
NAME	Rating Matrix	Application Vector	Intermediate Vector
SIZE	$m \times n$ m = # of features n = # of goals	$n \times 1$	$m \times 1$
INFORMATION CONTENT	R_{ij} = contribution of i^{th} feature toward j^{th} goal Language-independent Application-independent	A_j^U = weighting of j^{th} goal with respect to application area U Language-independent Application-dependent	IV_i^U = contribution of i^{th} feature to application area U Language-independent Application-dependent
ELEMENT SIZE	$0 \le R_{ij} \le 10$	$0.0 \le A_j^U \le 1.0$ $\sum_{j=1}^{n} A_j^U = 1.0$	$0 \le IV_i^U \le 10$ IV_*^U = sum of entries in IV^U
REFERENCES (in Ch. 2)	Paragraph I.4a, II.3 (Table I)	Paragraph I.4e, II.c	Paragraph I.4f, II.4 (Table II)

	$K(U)$ *	$(L^C$ *	$IV^U)$ →	t
NAME		Language Vector	Intermediate Vector	Technical Merit
SIZE	Scalar	$1 \times m$ m = # of features	$m \times 1$	Scalar
INFORMATION CONTENT	Transformation constant Language-independent Application-dependent	L_i^C = degree to which i^{th} feature is present in language C Language-dependent Application-independent	IV_i^U = contribution of i^{th} feature to application area U Language-independent Application-dependent	Technical merit of language C in application area U Language-dependent Application dependent
ELEMENT SIZE	$100/IV_*^U$	$0 \le L_i^C \le 1.0$	$0 \le IV_i^U \le 10$	$0 \le t \le 100$
REFERENCES (in Ch. 2)	Paragraph I.4g	Paragraph V.1,...,X.1 (Tables IV through IX)	Paragraph I.4f, II.4 (Table II)	Paragraph V.1,...,X.1

For example, the number of elements in the Rating matrix could easily be in the thousands.)

2. Summary of Technical Evaluation.

a. Determining Features and Goals. A preliminary step is the identification of (1) a spanning set of features found in HOLs, and (2) a complete set of goals that are relevant to HOL software.

b. Derivation of Rating Matrix. Once the features and goals have been determined, the derivation of entries in the Rating matrix R may proceed. The score R_{ij} is to reflect the degree to which the i^{th} feature contributes toward the j^{th} goal; this value is to be between 0 and 10, inclusive.

c. Derivation of Application Vector. Once the set of goals is determined, the Application vector A^U may be derived for application area U. The score A_j^U is the weighting of the j^{th} goal with respect to application area U. The scores in the application vector must be non-negative and sum to 1.0.

d. Derivation of Language Vector. After the set of features has been determined, the Language vector L^C may be derived for candidate language C. The score L_i^C is to reflect the degree to which the i^{th} feature is supported by language C; this value is to be between 0.0 and 1.0, inclusive.

e. Computation of Intermediate Vector. From the Rating matrix R and the Application vector A^U, the Intermediate vector $IV^U = R*A^U$ is computable. The score IV_i^U reflects the degree to which the i^{th} feature contributes toward application area U, and is in the range of 0 to +10.

f. Computation of Transformation Constant. The scalar value L^C*IV^U is between 0 and IV_*^U, inclusive, where IV_*^U is the sum of entries in IV^U. The constant K(U) is defined as $K(U) = 100/IV_*^U$.

g. Computation of Technical Merit. The scalar value $K(U)*(\overline{L^{C}*IV^{U}})$ is a result in the range 0 to 100 that reflects the technical merit of language C with respect to application area U.

3. Summary of Overall Evaluation.

a. Determining Management Decision-Making Criteria. The first step of the overall evaluation is the establishment of a set of criteria that will be used to govern the selection of a HOL. It is assumed that technical merit will be one of these criteria.

b. Derivation of Language Management Vector. After the set of management criteria is determined, the Language Management vector $LM^{C,U,E}$ may be derived for language C with respect to application area U and computing environment E. The score $LM^{C,U,E}$ is to reflect the merit of language C with respect to the k^{th} criterion, assuming that the HOL is used for application area U in computing environment E. Each such value is in the range 0 to 100.

c. Determining Relationships between Management Decision-Making Criteria and System Costs. After the set of management decision-making criteria is established, the mathematical functions that relate the scores for these criteria to actual system costs may be determined. This is likely to be the most difficult step to carry out.

d. Estimation of System Costs. Using the Language Management vector derived in b in conjunction with the relationships established in c yields an estimated value for the cost of implementing the given software system in candidate language C. A comparison of these estimates for the candidate HOLs will then reveal the cost-effectiveness of the various choices.

Section V. TACPOL AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for TACPOL's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

1. Technical Merit.

The language vector for TACPOL is displayed in Table IV. Multiplying the vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 264.9 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 49.

TABLE IV. LANGUAGE VECTOR FOR TACPOL

	1	2	3	4	5	6	7	8	9	10	11
A	.5	.6	.0	.7	.2	.6	.5				
B	.4	.4	.7	.9	.0	.7	.0	.5	.4	.5	.5
C	.0	.9	1.0	.0	.6	.3	.6	.0	.0		
D	.6	.9	.0	.8	.6	.0					
E	.0	.0	.9	.0	.0	.6	.5	.0			
F	.4	.4	1.0	.8	.8	.8	.7				
G	.7	.8	.8	.9	.1	.0	.1	.1			
H	1.0	1.0	1.0	.9	.5	.3	.8	1.0	.8	.7	
I	.2	.4	.0	.0	.8	.5	1.0				
J	.9	.0	.8	.6	.0						

2. Availability.

a. Compilers. Despite the fact that TACPOL's simplicity and orientation toward the AN/GYK-12 machine characteristics should permit the construction of good compilers, this area has been a source of problems for users of the language. Difficulties such as the generation of inefficient or incorrect code have resulted in heavier use of MOL in TACFIRE and TOS than would otherwise have been necessary. There has also been a minor but persistent problem with character encodings (specifically, conflicts with IBM's ASCII representations for "not" and "or").

b. Tools. There is room for improvement in the support tools provided for TACPOL. Desirable additions include better formatting and cross-referencing for output listings, a macro facility and a provision for inserting source code in a TACPOL program at compile-time, more extensive COMPOOL and library facilities, and better maintenance and diagnostic tools.

c. Documentation. The TACPOL documentation [3 and 6] is adequate, with [3] providing a formal definition and [6] serving more as a user's guide. The execution model presented in [3] is particularly helpful in describing the behavior of the various language constructs.

3. Standardization Status.

TACPOL does not satisfy very well the criterion of possessing a strongly enforced, formal standardization program; however, it has never been an objective to produce implementations of the language on a variety of machines.

4. Industry Support.

There is no support or use of the TACPOL language outside of the Army and Litton and other contractors working on Army tactical systems.

5. Availability of Trained Programmers.

Because of the simplicity of the language, training costs for TACPOL programmers should be relatively low; currently, a TACFIRE programming course at USACSC includes about a week and a half on TACPOL.

6. Effects of Criteria on Costs.

a. Language Acquisition. Costs in this area are relatively small, in view of the availability of the language in the Army. Compiler and support tool improvement are areas in which future expenses are likely.

b. Programmer Training. As mentioned in paragraph 5 above, this factor can be expected to be the smallest component of language-related cost.

c. System Development. The technical inadequacies in TACPOL and the failings with respect to compiler performance and support tools contribute heavily toward high costs in this area. The use of a technically superior language, together with a more complete and reliable language support system, could effect noticeable reductions of system development costs.

Section VI. CS-4 AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for CS-4's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

1. Technical Merit.

The language vector for CS-4 is displayed in Table V. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 410.2 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 76.

TABLE V. LANGUAGE VECTOR FOR CS-4

	1	2	3	4	5	6	7	8	9	10	11
A	1.0	.9	.4	.2	.0	.9	1.0				
B	.9	.9	.7	1.0	1.0	1.0	.4	.9	1.0	.9	.9
C	1.0	.9	1.0	.8	.2	.9	.6	.7	.0		
D	1.0	.9	.8	1.0	.9	.0					
E	.8	.3	1.0	.0	1.0	1.0	.8	1.0			
F	1.0	1.0	1.0	.7	1.0	1.0	.9				
G	.9	.9	.8	.8	.0	.9	.8	.6			
H	.8	1.0	.4	.9	.8	.8	.7	1.0	.1	1.0	
I	.4	.6	.3	.2	.3	.2	1.0				
J	.8	1.0	.9	.8	1.0						

2. Availability.

Since CS-4 has not been implemented, there are no compilers or support tools available. The reference manual and operating system interface description [4 and 5] are adequate for informal documentation, but a more complete and formal specification is needed.

3. Standardization Status.

There is no formal standardization program for CS-4.

4. Industry Support.

As an unimplemented language, CS-4 has no industry support.

5. Availability of Trained Programmers.

Again, because of the absence of implementations, CS-4 fails to meet this criterion.

6. Effects of Criteria on Costs.

a. Language Acquisition. The lack of implementations makes this cost factor significant. The effort required would be to polish the design, to implement the language and a set of support tools, and to provide user documentation.

b. Programmer Training. Once the language is implemented, expenses in this area should not be large.

c. System Development. The technical superiorities of CS-4 over the other five candidate HOLs suggest that the costs for system development will be relatively low if CS-4 is used.

Section VII. JOVIAL AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for JOVIAL's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

1. Technical Merit.

The language vector for JOVIAL is displayed in Table VI. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 274.0 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 51.

TABLE VI. LANGUAGE VECTOR FOR JOVIAL (J73/I)

	1	2	3	4	5	6	7	8	9	10	11
A	.5	.7	.4	.1	.0	.7	.4				
B	.6	.6	.9	.9	.6	.8	.0	.3	.3	.0	.7
C	.0	.8	1.0	.6	.6	.4	.6	.1	.0		
D	.3	.6	.4	.2	.7	.2					
E	.4	.0	1.0	.0	.0	.6	.5	.0			
F	.8	.6	1.0	.8	1.0	1.0	.0				
G	.6	.6	.8	.6	.1	.0	.1	.0			
H	.7	1.0	1.0	.7	.5	.7	.8	1.0	.0	.6	
I	.1	.6	1.0	.9	.8	.5	.8				
J	.9	.0	.3	.7	.0						

2. Availability.

JOVIAL (specifically, J73/I) is quite weak in this area. There are currently only two implemented compilers, and the language does not enjoy a very heavy usage. In fact, there is no usage of JOVIAL outside the Air Force. With respect to documentation, the reference manual [14] could be improved. The organization of this document was not clear, the BNF rules were presented neither top-down nor bottom-up, and the explanations in the semantics sections were not always complete.

3. Standardization Status.

Although JOVIAL might be expected to rate highly with respect to this criterion, in actual experience the standardization program has not prevented the development of numerous incompatible dialects (of previous versions of JOVIAL). The absence of I/O from [14] suggests that similar problems may occur in the future.

4. Industry Support.

One of the main failures of the JOVIAL effort in the past has been in precisely this area; there has never been a substantial amount of support (i.e., compiler and tool development) in industry.

5. Availability of Trained Programmers.

The number of JOVIAL programmers available to the Army is negligible, because of both the newness of the language and the unavailability of JOVIAL outside the Air Force.

6. Effects of Criteria on Costs.

a. Language Acquisition. The costs here involve the production of new code generators for existing JOVIAL compilers, to make the language available on Army hardware, and the development of language support tools.

b. Programmer Training. This expense should be relatively small, since JOVIAL is not a large language.

c. System Development. Costs in this area will likely be roughly comparable to those resulting from TACPOL usage. Reductions could be effected with the use of a technically superior HOL.

Section VIII. FORTRAN AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for FORTRAN's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

1. Technical Merit.

The language vector for FORTRAN is displayed in Table VII. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 222.7 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 42.

TABLE VII. LANGUAGE VECTOR FOR FORTRAN

	1	2	3	4	5	6	7	8	9	10	11
A	.4	.7	.1	.2	.1	.7	.1				
B	.3	.4	.7	.9	.8	.7	.0	.5	.3	.5	.5
C	.0	.9	1.0	.7	1.0	.5	.7	.0	.0		
D	.9	.7	.7	.0	.3	.0					
E	.1	.0	.8	.0	.0	.2	.5	.0			
F	.6	.3	.8	1.0	.0	.0	.5				
G	.4	.6	.4	.4	.0	.0	.3	.0			
H	.5	1.0	1.0	.9	.0	.5	.7	1.0	.9	1.0	
I	.0	.0	.0	.0	.8	.5	1.0				
J	.9	.2	.0	.0	.0						

2. Compiler, Support Tool, and Documentation Availability.

a. The widespread availability of FORTRAN has been one of the greatest advantages that FORTRAN has held over its competitors. The set of support tools developed is quite broad, ranging from software instrumentation packages to language preprocessors that enable users to code in more "structured" dialects. The documentation for the language has also been a strong point for FORTRAN; there are numerous textbooks that either teach the language or use it for specifying algorithms.

b. It should be pointed out, however, that the draft proposed FORTRAN [1] used for evaluation in this report does not possess any of these advantages at present, in view of the large differences between [1] and the previous standard.

3. Standardization Status.

FORTRAN has had a USANSI standard since 1966, but there are nearly as many different FORTRAN versions as there are compiler writers. Each version has its own set of extensions and restrictions. The result has been that, except for straightforward numerical programs doing no character processing or I/O, FORTRAN programs have not generally been portable. With the acceptance of the proposed draft standard [1], this situation should improve, because many operations that could be done only with "tricks" in earlier versions have become part of the language, and other operations have become better defined. Implementation-dependencies remain in the definition of floating-point precision, so that a calculation requiring only single precision on one machine may require double precision on another.

4. Industry Support.

The situation with respect to industry support is similar to that described above concerning availability: the experience with previous FORTRAN standards has been strong support throughout the computer industry, but the version of the language proposed in [1] is too recent to base a judgment upon.

5. Availability of Trained Programmers.

FORTRAN's widespread use in the technical and scientific communities and as a teaching language in

universities has resulted in a large supply of trained programmers, and this situation is also true in the Army. Again, however, this pertains to previous versions of FORTRAN, not [1]. The similarity between the languages facilitates retraining existing programmers, and the improvements introduced should lighten the task of teaching new programmers. However, it should also be noted that the teaching of FORTRAN in academic environments appears to be declining, as many universities are emphasizing languages that are somewhat better suited to the description of complex algorithms.

6. Effects of Criteria on Costs.

a. Language Acquisition. Because of the large differences between the draft proposed and the previous FORTRAN standard, acquiring the language defined by [1] entails implementation of new compilers and probably new support tools, as well as the production of new documentation. The costs involved would not be so large as the analogous expenses for CS-4, but they would still be significant.

b. Programmer Training. This expense should not be sizable.

c. System Development. Since FORTRAN is not as sound technically as other candidate HOLs considered in this report, system development costs are likely to be higher than for the other languages. We also note here that the absence of direct code disqualifies FORTRAN from consideration (unless the language is modified), since such a facility is an absolute requirement for the area of embedded computer applications.

Section IX. COBOL AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for COBOL's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

1. Technical Merit.

The language vector for COBOL is displayed in Table VIII. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 252.7 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 47.

TABLE VIII. LANGUAGE VECTOR FOR COBOL

	1	2	3	4	5	6	7	8	9	10	11
A	.4	.4	.0	1.0	.5	.6	.7				
B	.7	.6	.8	.9	.5	.6	.1	.0	.9	.9	.0
C	.7	.7	.0	.0	1.0	.0	.2	.0	.0		
D	.2	1.0	.4	.7	.6	.0					
E	.0	.0	1.0	.0	.0	.4	.5	.0			
F	.2	.0	.4	.5	.7	1.0	.8				
G	.4	.8	.8	.3	.0	.0	.5	.0			
H	.6	1.0	1.0	.9	.4	.5	.8	1.0	.0	.9	
I	1.0	.5	.0	.0	.1	.8	1.0				
J	.7	1.0	.6	.6	.0						

2. Availability.

COBOL is heavily supported in the Army computing environment; the only qualification is that the 1974 version of COBOL that was used as the basis of the evaluation has not been widely implemented.

3. Standardization Status.

One of the strongest points of COBOL is its history with respect to standardization. The committees responsible for maintaining the language since its inception in 1959 have been responsive to the needs of users (both in private industry and the government) and computer manufacturers, with respect to the language requirements for business applications. Appendix A in the COBOL reference manual supplies a brief history of the standardization effort.

4. Industry Support.

As mentioned in the preceding paragraph, the development of COBOL enjoyed from the start active involvement on the part of the industrial community. As Jean Sammet, one of the individuals who participated in the early work on COBOL, relates [17, p. 332]:

> The most significant aspect of this entire activity is that it was the first attempt...to have a group of competitors work together with the prime objective of developing a language that would be usable on computers from each of the manufacturers. That it succeeded is a tribute not only to the hard work of the individuals actually serving on the committee but more importantly to the management of the various companies involved who were able to recognize the value of subordinating their own individual plans and specialties to the broader overall benefit of the customers. This was particularly significant because many manufacturers were beginning or had already done considerable work on developing their own "commercial" languages and these developments naturally had to be eliminated or subordinated to the committee results.

5. Availability of Trained Programmers.

Because of the widespread use of the language, both within and outside the Army, COBOL meets this criterion fairly well. The only qualification (as with the availability criterion) is that the differences between the 1974 and the previous standard imply additional costs in this area. However, these differences are, for the most part, minor, and they are well documented in [8].

6. Effects of Criteria on Costs.

a. Language Acquisition. Because of the availability of COBOL, the costs in this area should not be significant.

b. Programmer Training. Expenses in this area should also be fairly small; however, the complexity of COBOL will cause these costs to be higher than with other of the candidate languages.

c. System Development. Costs here will be significantly higher than with other, more technically suitable HOLs. The problem is that COBOL is not well matched to application areas outside of business data processing. As a result, programming costs are likely to be considerably greater than they would be for HOLs that are technically superior.

Section X. PL/I AND THE MANAGEMENT DECISION-MAKING CRITERIA

This section presents the score for PL/I's technical merit and describes informally the language's compliance with the "non-technical" criteria pertinent to HOL selection.

1. Technical Merit.

The language vector for PL/I is displayed in Table IX. Multiplying this vector by the product of the Rating Matrix and the Application vector (see paragraph II.4) results in the scalar value 277.1 with respect to the possible range 0 through 541.4; the corresponding value in the range 0 to 100 (after being rounded to two significant figures) is 51.

TABLE IX. LANGUAGE VECTOR FOR PL/I

	1	2	3	4	5	6	7	8	9	10	11
A	.7	1.0	.6	.9	.7	.8	.6				
B	.8	.7	.9	1.0	.3	.6	.6	.3	.9	.9	.4
C	.0	.7	1.0	.4	.8	.4	.4	.5	.2		
D	.0	.7	.8	.4	.8	.5					
E	.5	.0	.0	.0	.0	.5	.5	.0			
F	.9	.6	.7	.8	.0	.0	.5				
G	.9	.4	.7	.8	.9	.0	.8	.0			
H	.8	1.0	.9	.8	.6	.5	.9	.5	.3	.7	
I	.1	.8	.0	.0	.0	.0	1.0				
J	.4	.0	.0	.8	.0						

2. Availability.

a. The complexity of PL/I will likely limit implementations of the full language to large machines; however, it is probable that within a few years many large manufacturers will have near-standard PL/I compilers and support tools. A number of manufacturers of smaller computers are producing compilers for PL/I subsets. With the acceptance of the full proposed PL/I standard, the PL/I standards committee is now considering standardizing a general-purpose subset of the full language. The availability of a standard subset definition will encourage manufacturers of smaller machines to produce compilers for this subset.

b. PL/I is a fairly well-documented language, with a variety of reference manuals, guides, and primers available. Although these are not applicable to the language as defined by [2], but rather to earlier versions, the differences are sufficiently minor so as not to make this a problem.

3. Standardization Status.

On 9 August 1976, the standard proposed for the full PL/I language was accepted by the American National Standards Institute. The standard gives a number of previously obscure operations well-defined meanings. The standards committee has been strongly supported by the computer manufacturers and by the user community.

4. Industry Support.

A major problem with industry support for PL/I has been the size and complexity of the language; a substantial commitment of resources is required to develop compilers or support tools. However, there is a fair amount of support for subsets and "PL/I-like" languages.

5. Availability of Trained Programmers.

Outside the Army, PL/I is a widely used language and rates highly with respect to programmer availability. The language is less widely used in the Army, but the similarities to TACPOL would reduce the training needed for personnel already skilled in TACPOL. It should be noted, however, that the size and complexity of PL/I make programmer training a highly non-trivial effort.

6. Effects of Criteria on Costs.

a. Language Acquisition. Since the differences between the previous PL/I standard and [2] are relatively small, the costs in this area should not be large.

b. Programmer Training. As noted in paragraph 5 above, the complexity of PL/I results in training costs that are higher than those anticipated for the other candidate HOLs.

c. System Development. The complexity of PL/I and some of the language's basic design features (e.g., the preponderance of implicit conversions) combine to add cost in the system development area. This is especially true in the maintenance stage of the system's life cycle. We also note here that the absence from [2] of provisions for direct code disqualifies PL/I from consideration (unless the language is modified), since such a facility is an absolute requirement for the area of embedded computer applications.

Section XI. SUMMARY OF CANDIDATE HOLS WITH RESPECT TO COST CONSIDERATIONS

1. Language Acquisition.

The estimated order of the candidate HOLs with respect to anticipated costs for language acquisition is (from low to high): TACPOL, COBOL, PL/I, JOVIAL, FORTRAN, and CS-4.

2. Programmer Training.

The estimated order for the candidate HOLs with respect to anticipated costs for programmer training is (from low to high): TACPOL, JOVIAL, FORTRAN, CS-4, COBOL, and PL/I.

3. System Development.

The estimated order for the candidate HOLs with respect to anticipated costs for system development is (from low to high): CS-4, PL/I*, JOVIAL, TACPOL, FORTRAN*, and COBOL. It should be noted that the difference between CS-4 and PL/I is expected to be significant, compared with the differences between PL/I and the remaining HOLs.

4. Conclusions.

Considering that over the period in which a common HOL would be used the costs associated with system development far outweigh the language acquisition and programmer training expenses, we are led to the conclusion that a technically superior language is preferable to an existing but inferior one. As a result, among the six languages considered in this report, CS-4 is recommended for selection. This conclusion will be seen to be supported by the detailed evaluations described in the remainder of this report.

---

* Assuming the inclusion of a direct code facility.

CHAPTER 3

TACPOL EVALUATION

Section I. LANGUAGE SUMMARY

1. Lexical Properties.

TACPOL is a free-format language based upon the ASCII character set. For purposes of token formation, a fifty-character subset is employed [3, p. 32]; characters outside this set may be used in special contexts such as comments and literal character strings. The space character is significant in separating tokens, and all keywords are reserved. Identifiers may be of arbitrary length, but only the first five and last three characters are significant. The underscore may be used as a "break" character in identifiers (but is not in the fifty-character set). The comment convention in TACPOL is that used in PL/I: comments are delimited by "/*" and "*/".

2. Data Types.

The data types available in TACPOL can be categorized according to Figure 2.

a. <u>Scalars</u>. The four scalar types are short numeric, long numeric, character string, and bit string. Type checking as it exists in TACPOL is enforced only for these types; e.g., it is illegal to assign a short numeric value to a long numeric quantity. (The term "quantity" is used with a fairly specific meaning in TACPOL. As stated in [3, p. 29], "A quantity is a unit of storage employed to hold (store) a value.")

(1) <u>Behavioral properties common to all scalar data types</u>.

(a) <u>Assignment</u> and <u>relations</u> are defined for each scalar data type, and for no other data type. Thus, if q is a quantity having one of the four scalar types, and v is a value of some type, then the assignment

```
q = v;
```

is legitimate if and only if v has the same type

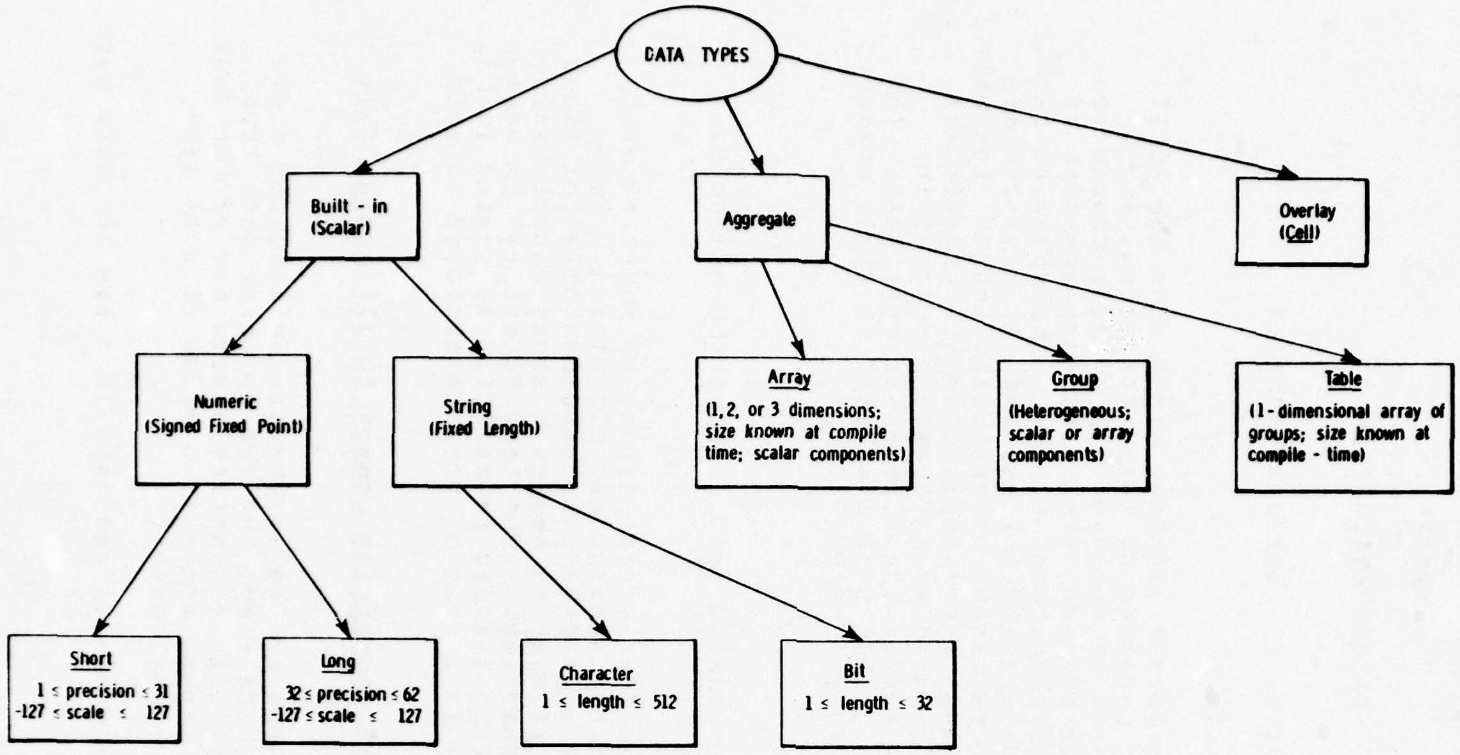

Figure 2. Data Types in TACPOL

as q. In the case of numeric operands, assignment may result in truncation of high-order and/or low-order bits, depending on precision and scale factor. For string operands, assignment when lengths differ will result in either truncation of rightmost characters or bits, or padding on the right with blanks or zeroes.

(b) The semantics of value parameters to procedures, of returning the results of function procedures, and of initialization of constants (in value declarations), are based on that of assignment.

(c) It may be noted that only scalars may be used as value parameters, function procedure results, or as constants.

(d) The relations which are defined for each scalar type are =, NE, GE, LE, GT, and LT. Though not explicitly stated in [3, Section 10.7.4], it appears that, for the numeric types, truncation may be performed during scale factor alignment. For string arguments, the shorter operand is padded on the right with blanks or zeroes before the comparison. The result of any relation is a bit string of length 1.

(2) Behavioral properties specific to the numeric types.

(a) The standard assortment of arithmetic operators are available in TACPOL. Unary + and - take a numeric operand and produce a value of the same type. Binary +, -, *, /, ** take two numeric operands (both short or both long) and produce a value of the same type. Exponentiation is restricted: only (non-negative) integer literals are allowed as exponents. Two additional operators allow the direct production of a long numeric value from short numeric operands: (*) and (**). The order of evaluation of operands in an expression is unspecified.

(b) Implicit scale factor and precision management are performed by the compiler for arithmetic

expressions. In the process, truncation of high-order and/or low-order bits may occur.

(c) A variety of utility routines (called "intrinsic procedures" in TACPOL) are defined on numeric data. These perform trigonometric and logarithmic functions, rescaling, etc.

(3) Behavioral properties specific to the string types.

(a) The CAT operator and the SUBSTR function apply to character or bit string data. The CAT operator produces the concatenation of the two operands (both of which must be either character strings or bit strings), truncating from the right end any characters (bits) which cause the result to be longer than 512 (32). The SUBSTR function (which produces a substring) may appear either on the left side or right side of an assignment. Its arguments are a string, an expression which evaluates to an index into the string, and a positive integer literal defining the length of the resulting substring.

(b) In addition to CAT and SUBSTR, there are several logical operations which apply only to bit string data (and not to character strings). These are OR, AND, and NOT; each produces a bit string result derived from a bit-by-bit computation on the argument(s) (the shorter operand is filled with 0's on the right before the operation). There is no short-circuiting of these operations; the order of evaluation is unspecified.

(4) Explicit conversion procedures. TACPOL's "redefinition procedures" allow a value of any scalar type to be converted to a value of any other scalar type. This may involve either an actual conversion process (e.g., long numeric to short numeric) or simply a reinterpretation of the existing bits without conversion (e.g., short numeric to character string or bit string).

b. Aggregate Data Types. TACPOL contains some facilities for defining homogeneous or heterogeneous data structures; viz., the array, group, and table generators.

(1) Properties common to all aggregate data types. Component selection is the only behavioral property common to all aggregate types; assignment and relations are not defineed for these types. A representational property of aggregate types is packing: the user may specify PACKED (to minimize data space) or ALIGNED (to minimize accessing time).

(2) Arrays. TACPOL permits the specification of one-, two-, and three-dimensional arrays of scalars; the size of each dimension is given by a positive integer literal appearing in the declaration. Subscripting forms are used to select contiguous sub-arrays or individual scalar components.

(3) Groups. A group is a heterogeneous data structure consisting of scalar or array components, each having a name. This name itself is used to select the corresponding component of the group.

(4) Tables. A table is a one-dimensional array of groups; the size of the table is given by a positive integer literal. Subscripting the table name yields one of the groups. Subscripting a group component name yields the corresponding component in the selected group.

c. Untyped Data. TACPOL contains several facilities which defeat the type checking for scalar data, and the language contains no features for enforcing type checking with aggregate data.

(1) Treating data as a bitstream. The intrinsic procedures MOVE and CLEAR [3, p. 89] perform bit-transfering functions independent of the types of their arguments. MOVE(x,y) simply copies x into y as far as the shorter of the two; CLEAR(x) puts all Ø's into x.

(2) Overlaying data objects. The CELL declaration has the effect of a free union; it permits instances

of scalar or aggregate types to be overlaid on the same storage area. Quantity parameter passing (i.e., "by reference") has a similar behavior. If q is a quantity passed as an argument to a procedure, where the corresponding formal parameter is a quantity parameter p, then during the execution of the procedure the effect will be that of overlaying p on the storage required for q, regardless of the data types of p and q.

3. Procedures.

a. Proper and Function Procedures. TACPOL allows the definition of "proper procedures," which do not return results; these are used in CALL statements. In addition, "function procedures" are permitted; these return scalar values as results and are used in expressions.

b. Formal Parameters. For either proper or function procedures, there are four categories of formal parameters.

(1) Value parameter. This has the effect of a "copy-in" parameter; only scalar data types are permitted. The formal parameter can be used as a local variable in the procedure (i.e., it may be assigned to).

(2) Quantity parameter. As mentioned above, this is analogous to the "by-reference" parameter in many languages, except that no type checking is performed to match the argument with the formal parameter.

(3) Procedure parameter. Parameterless proper procedures may be passed as arguments to other procedures.

(4) Point parameter. Points (i.e., statement labels) may be passed as arguments to procedures.

c. Recursion. Recursion is apparently permitted, but only for procedures which are programs.

4. Statements.

a. Null Statement. This statement allows extra semicolons to appear in programs.

b. Assign Statement. The semantics of assignment was described above (in sub-paragraph 2a(1)).

c. Call Statement. This statement is used to invoke proper procedures.

d. GoTo Statement. The TACPOL GoTo statement may be used to transfer control to any label whose name is known in the scope of the statement. Thus transfers out of procedures are permitted, but jumps into loops are not.

e. Conditional Statements.

(1) If statement. The TACPOL if-statement may have either one alternative (following THEN) or two alternatives (THEN and ELSE). The existence of one or more 1-bits in the bit-expression following IF is regarded as the "true" condition.

(2) Switch procedure. The effect of a CASE statement (or computed goto) can be obtained in TACPOL by calling an intrinsic procedure named SWITCH.

f. DO Statement. The TACPOL DO statement allows iterative execution of a sequence of statements. Two varieties of iteration are possible: stepping a loop variable by fixed amounts until a terminal value is reached, or executing the statements as long as some condition (a bit-expression) is "true". The DO statement will include one or both of these varieties. If a loop variable is used, it will receive an implicit declaration of BIN FIXED(15,0) and will thus be local to the DO statement; in addition, it may not be assigned to by the programmer.

g. Blocks. A sequence of declarations and statements may be bracketed by BEGIN and END to form a block, which may then be used as a single statement.

h. Code Statement. TACPOL allows the user to insert non-TACPOL code in the program; this is achieved via the code-statement.

5. Storage Allocation.

Storage allocation in TACPOL can be carried out statically, since the size of each data object is known at compile-time and since recursion inside programs is prohibited.

6. Process Scheduling.

TACPOL contains no facilities for process scheduling.

7. Files and I/O

a. TACPOL contains a variety of features for dealing with data files. Files may be _partitioned_ or _nonpartitioned_; each partition in a partitioned file is accessed by a unique character string key. In addition, files (partitions) may be either _serial_ (sequential access) or _direct_ (random access).

b. There are three modes of access to a file: _input_, _output_, and _update_. Only direct files are permitted to be accessed for update.

c. Concurrency of program execution with data transmission during file processing is allowed in TACPOL; the default behavior, though, is to wait for the data transmission to be completed.

d. The file processing statements include OPEN, CLOSE, READ, WRITE, DELETE, SPACE, REWIND, UNWIND, and several others. File contents in TACPOL are presumed to be unstructured, binary records. No formatted I/O is available. No data type information is preserved with a file, making it possible to WRITE data of one type and to READ it back as another.

e. TACPOL provides an On-statement which allows conditional execution based on whether an indicated file condition (e.g., reading an end-of-file, accessing a record with an illegal key) occurred during a previous file processing operation.

8. Exception Handling.

Very little information is provided in [3] concerning error detection, either at compile-time or run-time. In general, erroneously obtained values (e.g., due to fixed-point overflow) are said to be "undefined", the implication being that no run-time checks are made. If it is desired to obtain such checks, condition declarations may be used to explicitly indicate this. Conditions which may be declared include zero divide, fixed overflow, assignment to a user-specified variable, and invocation of a procedure in a user-provided list. If any of the declared conditions occurs, a "snap" procedure (not defined in [3], but implied to be a trace of the block in which the condition occurred) is executed.

9. Compile-Time Facilities.

a. The TACPOL compool facility is described briefly in [6] (but not in [3]). A compool generator accepts TACPOL declarations (e.g., procedures, files, quantities) and outputs binary and symbolic data which are used as input to the TACPOL compiler. A compool serves as an outermost namescope for the program which includes it.

b. TACPOL has no facilities for macros, conditional compilation, or for the inclusion of separately written source text. A separate compilation facility is implicit in [3] but is not defined there.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. Type checking in TACPOL exists only for scalar data (short and long numeric, character and bit string), and only in the contexts of expressions and assignments. No type checking is performed when quantity (i.e., "by reference") parameters are passed. The MOVE and CLEAR intrinsic procedures treat data as untyped. The CELL facility for overlaying data objects allows type checking to be easily defeated, since no provision is made for determining which alternative is in use at run time.

c. TACPOL allows implicit declarations in the case of loop control variables, in violation of A1.

d. The changes needed to bring TACPOL into compliance with Tinman requirement A1 are fairly major, and it is doubtful whether the language could be so modified and still keep its essential character. The language facilities impacted are the data definition features, assignment, relational operators, and procedures.

e. First, if specifications of arrays, groups, or tables are to be interpreted as data types, the issue of type identity must be faced. For consistency with the scalars, the size of arrays or tables should not distinguish data type, nor should the packing (i.e., ALIGNED vs. PACKED). Groups present a problem: the simplest rule for type identity is for the field names to match and for the corresponding component types to be the same. Unfortunately, TACPOL treats field names as declared quantities and thus prohibits two groups with common field names from being specified at the same level in a namescope.

f. The assignment statement and comparison operators (for equality and inequality) should be extended to deal with data of any data type, not just scalars. The type checking would ensure, e.g., that a table was not assigned to a group.

g. Quantity Parameter passing must be made more restrictive in the interest of program reliability. It is

safe to pass an argument to a quantity parameter only if their types are the same and they are represented in exactly the same way. Thus scaling and precision (for numeric data), size (for strings, arrays, tables), and packing are relevant attributes which should be checked.

h. Implications of providing a discriminated CELL feature are discussed in A7 below.

2. Data Types (A2).

a. Degree of compliance: P

b. First, there is no floating point data type. Second, there is no explicit Boolean type. Boolean is a special case of a bit string (length 1), a situation which is not always desirable: e.g., AND and OR should not be short circuited for bit string, but they should be short circuited for Boolean (see also B6). Third, the array, group, and table facilities are restricted as to the number of dimensions and the attributes of their components (e.g., a group cannot be a component of another group).

c. The changes implied by A2 are non-trivial. If the language is to incorporate floating point, then the problem of differentiating between fixed and floating point literals arises. Also, the issue of mixed mode arithmetic must be faced; if expressions may contain both fixed and floating point operands, then conversion rules must be stated. Introducing a separate type for Boolean may be tricky, since the relationships between it and BIT(1) can be subtle. For example, if no separate literal form is provided, then the interpretation of, say, '1'B is ambiguous. Generalizing the array, group, and table features will probably necessitate a different set of rules for accessing components, and a modified syntax for declaring data objects.

3. Precision (A3).

a. Degree of compliance: F

b. Lacking a data-type for floating point values, TACPOL fails to satisfy this requirement. The impact of adding floating point was described in conjunction with A2 above.

Under the assumption that floating point is added, the additional modifications necessary to bring TACPOL into compliance with A3 are not major, but they would cause an inconsistency between the declarations for fixed and floating point quantities. (There is no requirement in the Tinman that global specification of precision for fixed point arithmetic be supported.)

4. Fixed Point Numbers (A4).

a. Degree of compliance: P

b. TACPOL fixed point numbers are binary (and thus approximations to decimal values) instead of exact as required by A4.

c. The changes needed to bring TACPOL into compliance with A4 are relatively minor; the syntax should be changed from BIN FIXED(precision, scaling), and the interpretation of literals and the results of arithmetic operations should be modified appropriately. It should be pointed out, though, that such changes have several undesirable effects. First, run-time efficiency is impaired, since scale factor alignment will (in the absence of decimal hardware) be implemented through multiplication by powers of 10, instead of simply binary shifting. Second, the rules governing literals will be made more complicated, in view of the fact that short and long numeric are distinct types in TACPOL. For example, some literals with (decimal) precision 10 will be short numeric, while others will be long.

5. Character Data (A5).

a. Degree of compliance: P

b. A5 requires that character sets be treated as any other enumeration type, and TACPOL does not contain features for defining enumeration types. However, the relational operations are defined for character data, so that some of the effects of treating character sets as enumeration types are in fact achieved.

c. The modifications needed so that enumeration types can be defined are described in connection with E6 below.

It should be pointed out, though, that attempting to obtain the character data type by such a facility is not likely to be cleanly achievable. One problem is that a single character currently is obtainable as a character string of length 1, but there is no corresponding concept of a string of objects from an enumeration type. Associated with this problem is the issue of how a character string literal can be defined, since in general there is no literal form for arrays of enumeration types.

6. Arrays (A6).

a. Degree of compliance: P

b. In TACPOL, the lower subscript bound is always 1, and the upper subscript bound must be a positive integer literal (thus known at compile time).

c. Allowing run-time (scope entry) determination of array bounds is a major change to TACPOL, necessitating a redesign of much of the data-type facility (e.g., for consistency the sizes of bit strings and character strings should also be run-time determinable). Such a change, however, cannot be carried out without defeating many of the essential objectives of TACPOL (e.g., simplicity, run-time efficiency, compile-time determination of all addresses and storage requirements).

7. Records (A7).

a. Degree of compliance: F

b. Although the CELL feature allows records of alternative structure [3, pp. 42-44], this is accomplished via a simple storage overlay technique which defeats type checking. In addition, the CELL facility restricts the kinds of data objects which may be overlaid; for example, one may not overlay a table on two contiguous groups.

c. To bring the language into compliance with A7, it would be necessary to include a "tag" field with each CELL object. A run-time check would then be generated for each reference to a CELL component, to guarantee that the current value of the tag corresponds to the referenced component.

This is a non-trivial change, and, to prevent circumventions of type-checking, impacts the procedure facility. (If quantity passing of CELL components is allowed, it is possible to obtain distinct names, with distinct data types, for the same storage.) In addition, the (implicit) generation of run-time checks is antithetical to TACPOL's goal of object code efficiency.

Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. TACPOL partially satisfies B1, with the following exceptions. First, assignment is defined only for scalar types -- one cannot, for example, assign an array to another array. Second, since there are no facilities for encapsulating type definitions, there are no user-definable assignment or accessing operations. Third, since the CELL feature realizes storage overlaying, reference to a quantity will not necessarily retrieve the last value assigned to that quantity.

c. Major modifications are required to bring TACPOL into compliance with B1. Extending assignment to non-scalar data was described above in connection with A1. The encapsulation of type definition is discussed below (E5). Implications of a "safe" CELL facility were presented above in connection with A7.

2. Equivalence (B2).

a. Degree of compliance: P

b. The identity operation in TACPOL is applicable only to scalar data and is legal only if both operands have the same type; thus, TACPOL is not as general as required in B2.

c. The scope of modifications is roughly the same as that required to extend assignment to non-scalar data. This would not be a major change to the language.

3. Relationals (B3).

a. Degree of compliance: P

b. All six relational operators are in fact automatically provided for numeric data (i.e., short and long numeric), but there are no enumeration types in TACPOL. Also, shifting due to scale factor alignment may result in the loss of significant bits with no indication of error.

c. The only modifications needed are those which bring enumerated types into the language; these are discussed in E6 below.

4. Arithmetic Operations (B4).

a. Degree of compliance: P

b. Since there is no floating-point data type, the Tinman requirement that division yield a real result is not satisfied. Exponentiation is restricted in that the exponent must be a non-negative integer literal.

c. The impact of introducing a floating-point data type was described above (A2). The generalization of exponentiation is a relatively minor change but necessitates a set of special cases in the language specification.

5. Truncation and Rounding (B5).

a. Degree of compliance: F

b. TACPOL fails to satisfy this requirement. Depending on scale factor and precision, the most significant bits may be truncated during assignment, arithmetic operations, and relational operations.

c. Bringing the language into compliance with B5 represents a major change with respect to TACPOL's design goals, since it would imply run-time checking which can be prohibitively expensive in the absence of suitable hardware. This problem could perhaps be circumvented by providing facilities which allows the user to suppress the generation of the checking code, but this would add complexity to the language, again defeating one of TACPOL's goals.

d. Although the modifications needed to make TACPOL comply with B5 are significant with respect to the flavor of the language, the effects should be fairly localized with respect to the compiler. (Presumably, only the code generator phase would be affected.) Thus, the impact of the modification on an implementation is not major.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. The operators AND, OR, and NOT are actually provided for arbitrary bit strings, and not simply for BIT(1). There is no NOR operator available. There is no "short-circuiting" of the AND and OR operators.

c. Introducing a NOR operator is a minor change. Providing a "short-circuiting" for AND and OR on arbitrary bit strings is undesirable, but to effect this behavior for BIT(1) and have other rules for BIT(n) where n > 1 is apt to be confusing. Alternatively, providing a separate Boolean data type, distinct from BIT(1), also raises some difficulties (see A2 above). Such problems make the introduction of short-circuiting a non-trivial design change.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. TACPOL does not permit scalar operations or assignment to be performed on arrays, groups, or tables.

c. A major design change is required to bring TACPOL into agreement with this Tinman feature. The issue of extending assignment to arrays, groups, or tables was described earlier (A1); note that B7 also implies that scalar-to-array assignment be permitted. The extension of scalar operations (e.g., +, -, *, /) to composite data structures implies a data definition facility not available in TACPOL; see E5 below.

8. Type Conversion (B8).

a. Degree of compliance: P

b. TACPOL satisfies this requirement in several ways: first, the "no implicit type conversions" requirement is enforced for the operations on scalar data; second, the language provides a complete set of intrinsic procedures (called "redefinition procedures") for explicitly converting data from one scalar type to another; and third, no conversion operation is required when the type of an actual

parameter is a constituent of a union type (i.e., CELL) which is the formal parameter.

c. However, there are also major ways in which TACPOL fails to satisfy this Tinman requirement. Most importantly, an implicit type conversion of a notorious variety -- interpreting storage which contains a value of one data type as though its contents are of some other data type -- which makes program maintenance exceptionally difficult, is built into the language in its quantity parameter passage and CELL facilities. A minor instance of implicit conversion occurs in array subscripting; fractional bits are truncated to ensure integer subscripts. It should also be pointed out that the explicit conversion between numeric and character string data is not likely to be the one desired by the user. For example, the string-to-short-numeric conversion SHORT('9') produces the BIN FIXED(8, 0) value 00111001 (Base 2) (the ASCII encoding of the character '9') instead of 00001001 (Base 2) (the value 9).

d. The major design changes required to bring TACPOL into agreement with B8 have been discussed earlier, in connection with A1.

9. Change in Numeric Representation (B9)

a. Degree of compliance: P

b. The first requirement of B9 ("Explicit conversion operations will not be required between numerical ranges") is partially satisfied: such conversions are not required when source and target are both short or both long, but they are required when one is long and the other is short.

c. TACPOL fails completely with respect to the second part of B9 ("There will be a runtime exception condition when any integer or fixed point value is truncated"). No such exception condition exists: TACPOL specifies that the result of an operation is "undefined" when such truncation (of high order bits) occurs [3, p. 59].

d. To satisfy cleanly the first requirement of B9, TACPOL should not distinguish at the language level between short and long numeric data types. This is not a major design change.

e. With respect to the second part of B9, the main modification to the language would be to make explicit that "undefined" results in fact are errors which are checked at run-time. The impact of such a change, on design philosophy and compiler implementation was discussed above in connection with B5.

10. I/O Operations (B10).

a. Degree of compliance: P

b. TACPOL contains facilities for dealing with serial and direct-access files comprising unstructured binary records. No features are available (without dropping into MOL) for handling channels or devices. No dynamic assignment of I/O devices is possible. A limited capability exists for user handling of exception conditions (e.g., reading end of file, attempting to access a record with an illegal key). No data type information is associated with files; it is possible to write information of one type and read it back as another type.

c. No formatted I/O is available in the language; explicit editing procedures are necessary.

d. The language places restrictions on the allowable sources and destinations for READ and WRITE which make it illegal to READ a record into, or WRITE a record from, a quantity specified by subscripting an array; it is also illegal to WRITE a character string literal.

e. The implication of B10 is that files, channels, and devices should be definable as data types via the language's type encapsulaion facility. TACPOL lacks such a facility; it is a major design change to introduce this into the language (see E1 below). The other changes needed to bring TACPOL into compliance with B10 are less far-reaching.

11. Power Set Operations (B11).

a. Degree of compliance: F

b. TACPOL does not contain or permit the definition of enumeration types; thus, it also does not provide for power sets of these types. However, using the bit string data type the programmer can simulate part of the behavior of power sets.

c. The main change needed to bring TACPOL into compliance with B1 is to allow enumeration types in the language (see E6 below); the provision of power set types is then not a major problem. However, the resultant redundancy between power sets and bit strings is not especially desirable.

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: F

b. TACPOL explicitly leaves undefined the order in which arguments of an expression (or arguments to a procedure) will be evaluated [3, Sections 10.7.1 and 10.4.8h].

c. It is a relatively minor and localized change to the language to define a left-to-right order for the evaluation of arguments producing side-effects. On the other hand, such a change has a large impact on the compiler if the intention is to preserve the object code efficiency which is facilitated by undefined order of evaluation.

2. Operand Structure (C2).

a. Degree of compliance: PT.

b. As evidenced in its syntax for expressions, TACPOL conforms almost completely with Tinman requirement C2; the exceptions are the following:

(1) In TACPOL, unary operators + and - have higher precedence than exponentiation. As a result, -5**2 would be interpreted as (-5)**2. A more natural interpretation is -(5**2).

(2) The concatenation operator (CAT) has low priority with respect to the other string operators. Thus '11'B CAT '011'B OR '10101'B is equivalent to '11'B CAT ('011'B OR '10101'B). Since such an interpretation is not "obvious to the reader" nor "widely recognized," explicit parenthesization should be required.

c. The modifications needed are minor. However, changing the precedence of operators in a language is generally inadvisable from a program maintenance viewpoint.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. In any context where both constants and references to variables of a given type are allowed, TACPOL also permits expressions of that type to appear. For example, there is no restriction on the form of subscripts for tables or arrays (any short expression is allowed).

c. Since C3 is independent of the other Tinman characteristics, satisfying C3 does not cause a conflict with other requirements.

4. Constant Expressions (C4).

a. Degree of compliance: F

b. TACPOL contains many features which require constants: e.g., dimension sizes for arrays and tables, precision and scaling for numeric data, string length specification, repetition factors preceding string literals, initialization in value declarations, exponent operand in exponentation, element count argument to SUBSTR. In none of these cases are constant expressions permitted.

c. The provision of a facility for interpreting constant expressions before run-time has direct impact on compiler costs, but does not affect the language design in a major way. It should be noted that compile-time (as opposed to load-time) evaluation is implied, since the constant expression value is needed to determine data storage requirements and/or object code.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement, with the following special cases:

(1) Only scalar data types are permitted as VALUE parameters.

(2) Only parameterless proper procedures can be passed as arguments.

(3) Only "quantity designators" are permitted to be passed as arguments to quantity parameters or to be used in READ or WRITE statements, which make it illegal to READ, WRITE, or pass an array component as a quantity argument, or to WRITE a literal value.

(4) "Only quantity and value arguments can be arguments to a procedure which is a program" [3, p. 47].

c. Allowing arbitrary data types for VALUE parameters entails extending assignment to non-scalar data; this also falls under the modifications described above with respect to A1.

d. Permitting arbitrary procedures to be passed as arguments will make the language and compiler somewhat more complex, but does not impact other features in the language.

e. Allowing subscripted quantities to be passed as quantity arguments is a minor change; similarly, generalizing the data source in WRITE statements to be expressions is a minor modification.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. TACPOL satisfies this requirement only to a limited extent. Type checking is enforced for value parameters, which follow the rules for assignment, but is completely lacking for quantity parameters. An additional discrepancy with C6 is that in TACPOL the size and subscript range for array parameters must be known at compile-time instead of being run-time computable.

c. Extending type-checking to quantity parameters, and permitting run-time determination of array sizes are both major changes to the language and the implementation; their implications were described above in connection with A1 and A6.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. The "constant" form of parameter, described in C7, is partially matched by TACPOL's value parameter passing. The differences are that TACPOL restricts such parameters to scalar data and allows the formal parameter to be modified (like a local variable) in the body of the procedure.

c. The "renaming" form of parameter corresponds closely to TACPOL's quantity parameter (except for the latter's absence of type checking). The inability to pass array components as quantity parameters is a special case in the language which seems to serve no useful purpose. Also, having quantity parameters (instead of value parameters) as the default appears to be the wrong choice, since hard-to-detect errors can result if the programmer is not conscious of this default.

d. TACPOL allows statement labels (termed "points") and parameterless proper procedures to be passed as arguments. A more general facility for passing procedures is required for C7.

e. The Tinman requirement that "exception handling control parameters will be specified on the call side only when needed" is not met. There is no facility in TACPOL for having optional parameters.

f. Most of the larger modifications needed to bring TACPOL's parameter passing facility into compliance with C7 have been described above. E.g., extending VALUE parameters to non-scalar data in a fashion consistent with the language implies a generalized assignment operation (A1, B1); type checking on quantity parameters was discussed in connection with A1. The provision for a more general procedure-parameter passing facility adds complication to the language but represents a fairly localized change; it is assumed that (for type checking purposes) the types of the parameters and (of a function) the result type will be required to be specified for procedures which are formal parameters. The Tinman requirement for "a formal parameter class specifying the control action when exception conditions occur" is unclear.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: P

b. Specification of all attributes of formal parameters is always required (i.e., is never optional). TACPOL contains no generic procedure facility of the variety envisioned in C8.

c. The capability described in C8 appears basically to be a text substitution macro facility. The addition of such a capability to TACPOL has minor impact on the rest of the language and has a localized but non-trivial effect on implementation.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. In order to support such a facility, TACPOL requires a more general array generator (i.e., components should be able to have arbitrary type and not just scalar). Also, procedures must be able to take array parameters whose size may vary. Type checking during parameter passing is implicit in C9; implications of type checking for quantity parameters were described above (A1). In summary, the language and implementation modifications needed to bring TACPOL into compliance with C9 are non-trivial.

Section V: VARIABLES, LITERALS, AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The INIT attribute (*) in value declarations [3, Section 10.4.7] is used for declaring constants, but applies only to scalars. Also, since the values specified are limited to literals, one cannot declare numeric constants whose values are negative (literals do not include a sign).

c. The language changes implied by this requirement are the introduction of a facility for compile-time evaluation of expressions, and the generalization of assignment to operate on non-scalar data. The former was discussed above in connection with C4, and the latter was described under A1.

2. Numeric Literals (D2).

a. Degree of compliance: PT

b. As required by D2, TACPOL provides literal forms for all the built-in data types. However, there are several problems with the way the literals are realized. One is a notational inconvenience: bitstring literals must be written in binary, whereas octal and/or hexadecimal would be more readable. Another problem (unavoidable in the presence of binary fixed-point) is the mixture of binary and decimal concepts in numeric literals [3, Section 10.5.1], which is likely to cause programmer confusion.

c. It is a trivial change to add octal and/or hexadecimal literals to TACPOL; a notation illustrated by '716'O or '5B4'X is sufficient.

3. Initial Values of Variables (D3).

a. Degree of compliance: F

(*) "INIT" is an unfortunate keyword to use in this context, since it suggests initial (instead of constant) values.

b. TACPOL has no provision for specifying initial values with variables as part of their declarations; as stated in [3, p. 30], "At the time that a quantity acquires an identity, it also acquires an undefined value."

c. The language changes needed to bring TACPOL into compliance with D3 are as follows. First, the INIT attribute should be changed in semantics so that it is used for initializations instead of constant definitions. Second, forms should be introduced for the construction of data aggregates (arrays, tables, groups, cells); these forms will be usable in initial value specification. Third, assignment with non-scalar data should be permitted, so that the rules are consistent whether scalar or non-scalar objects are initialized. These represent moderate changes to the language.

d. The impact on implementation is major if run-time testing for initialization is to be carried out. One possible technique is to have distinct modes of program compilation ("debugging" vs. "production"). A program compiled in "debugging" mode would have each of its data objects represented with an initialization tag (perhaps a single bit) which could be interrogated on each reference. The difficulty with such a scheme, however, is that it interacts unfavorably with separate compilation; handling mixtures of "debugging" and "production" modules can be non-trivial.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. Since precision and scaling are provided (either explicitly or by default) for each numeric type specification, the range and step size are derivable from these values. However, ranges are not arbitrary (*) ; with precision p and scale factor s,

----

(*) It should be pointed out that there is a contradiction between TACPOL references [3] and [6] concerning the sign of numeric data. It is stated in [3, p. 28] that "all numeric values are signed." However, [6, p. 2-1] asserts: "All numeric values with eight or more bits of precision are signed. All numeric values with seven or fewer bits of precision are unsigned."

the range is -(2**p - 1)/(2**s) through +(2**p - 1)/(2**s). Also, the Tinman provision that "range and step size specifications will not be interpreted as defining new types" is not satisfied by TACPOL; short and long numeric are distinct types which differ solely in the size of the precision. Moreover, the utilization of range specifications in TACPOL for degugging purposes is fairly limited, since implicit truncation is the rule when out-of-range values are assigned.

c. The changes needed to bring TACPOL into compliance with D4 vary in scope. The addition of arbitrary ranges, with run-time bounds checking, does not have major impact on implementation but does change the flavor of the language; similar remarks hold regarding the elimination of implicit truncation (see also B5 above). Removing the distinction between short and long numeric types, on the other hand, has minor impact on the language but causes non-trivial implementation difficulties. The problem which arises is that data objects of the same type would have different representations, thereby complicating such facilities as quantity parameter passing.

5. Variable Types (D5).

a. Degree of compliance: P

b. TACPOL contains a variety of restrictions which prevent it from satisfying D5. The components of arrays may only be scalars, and arrays are limited to three dimensions. Also, group and table components may only be scalars or arrays, limiting the forms of data structures directly definable in the language. In addition, the cell feature has restrictions which prevent it from supplying a general overlay facility (see A7 above). The absence of enumeration types also detracts from TACPOL's compliance with D5.

c. The main changes needed to bring TACPOL into compliance with this requirement are the introduction of enumeration types (see E6) and the replacement of the aggregate data structuring mechanisms with a more general facility. Instead of the current array, group, and table features, the language should provide homogeneous and heterogeneous structuring facilities which can take components of unrestricted type. Thus, array and group would be generalized, and there would be no need for a table

facility (tables are simply arrays of groups). These changes have a major impact on implementation but do not cause adverse interactions with other language features.

6. Pointer Variables (D6).

a. Degree of compliance: F

b. TACPOL contains no facility for defining pointers. The changes needed to bring TACPOL into compliance with D6 are major, with respect to both language and implementation. In order for a pointer facility to be "safe," user-defined data types are needed (perhaps not of the magnitude implied by E1, but at least an ability to associate type names with specifications of data types). Such a facility requires a considerable amount of design work. Also in connection with safety, a system-maintained garbage collector is needed; this has adverse effects on run-time efficiency and would conflict seriously with one of TACPOL's major design goals.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: F

b. TACPOL lacks the facilities needed to allow user definition of new data types and operations. The only definitional features in TACPOL are (1) a facility to define certain kinds of structured data objects (arrays, groups, tables), and (2) an ability to define callable procedures. These are not sufficient to meet the requirements of E1. Implicit in the notion of data type are restrictions on the uses of its member objects, but in TACPOL the lack of type checking for quantity parameters results in a complete freedom to use a data object in ways which ignore the object's declared structure.

c. It is unrealistic to modify TACPOL to bring it into compliance with E1, since the requirements listed can be satisfied only if the language designers make a conscious effort to include them as goals from the start. A data definition facility is a central component of any language, and has interactions with almost every other feature provided. Operator extension is somewhat more localized with respect to language design, but its incorporation would be antithetical to the simple nature of TACPOL and would complicate implementations.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. Lacking user-defined types, TACPOL fails to meet this requirement. Moreover, TACPOL shows inconsistencies between the uses of scalar and non-scalar data types. For example, assignment and equivalence are defined only for scalars; constants, value parameters and the returned values for procedures (based on the rules for assignment) are similarly restricted to be scalars.

c. As mentioned in connection with E1, major modifications, entailing a substantial redesign effort, are needed if a data type definition facility having the scope called for in the Tinman is to be provided. On the other hand, the revisions needed to make scalar and non-scalar

types consistent are somewhat less drastic; the main change is the extension of assignment to non-scalar data (described in A1 above).

3. No Default Declarations (E3).

a. Degree of compliance: PT

b. TACPOL basically satisfies this requirement; the only exception is the loop control variable of the DO statement, which has the default attributes BIN FIXED(15, Ø) [3, Section 1Ø.8.6].

c. It is a minor change, both to language and implementation, to require the specification of attributes for the loop control variable. However, this change may adversely affect efficiency, since programmer-specified attributes may prevent the holding of the variable in a register.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. TACPOL has no facilities for defining new data types and hence does not permit extension of operators to such types. Analogously to E1 above, there is no way to fulfill this requirement without drastic changes to the structure of the language.

5. Type Definitions (E5).

a. Degree of compliance: F

b. TACPOL does not permit the definition of new data types as required in E5. Again, as mentioned in connection with E1, such a facility cannot reasonably be added to a language unless the original design had made provisions for it.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: P

b. TACPOL partially fulfills this requirement, allowing data definition via Cartesian product (array, group, table)

and implementing power sets via bitstrings. However, the language fails to provide definition via enumeration of literal names, and it realizes storage overlaying as opposed to discriminated union.

c. The addition of enumeration types to TACPOL does not represent a major change to the language; however, there are a varity of subtle issues which must be faced which impact language and implementation. For example, whether the symbolic names comprising an enumeration list must be unique is a decision which presents a tradeoff between user convenience and language/implementation complexity. In addition, the meaning of the concept of data type in connection with enumerations is somewhat cloudy.

d. The issue of adding discriminated union was discussed in connection with A7 above.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. TACPOL partially fulfills this requirement. Consistent with E7, the language does not allow subset types. However, TACPOL does contain free union in the form of the cell facility and (in effect) quantity parameter passing.

c. Removing free union entails provision of a discriminated union facility and revision of the quantity parameter passing rules. The impact of the former was described in connection with A7; the latter was discussed in A1.

8. Type Initialization (E8).

a. Degree of compliance: F

b. TACPOL has no facilities for defining new types and hence lacks provision for user specification of type-specific initialization routines. As described in connection with E1, the basic structure of the TACPOL language is inconsistent with the data definition mechanism envisioned in the Tinman requirements, and it is unrealistic to attempt to modify TACPOL into compliance.

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1)

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The language provides block structure, and an object may be allocated but inaccessible within a block because its name has been redeclared there. However, there is no facility in TACPOL like "own" variables of ALGOL 60, in which an object is allocated in an outer block and accessible only within an inner block.

c. It is a relatively minor change, both to the language and to the implementation, to introduce such an "own" variable facility.

2. Limiting Access Scope (F2).

a. Degree of compliance: F

b. Lacking encapsulated type definition, TACPOL fails to satisfy this requirement with respect to limited access in a type definition. The issue of separate compilation is not directly addressed in [3], although the load statement [3, p. 82] and the references to COMPOOL [3, p. 73, pp. 125ff] imply that some such facility is available.

c. The changes implied by introducing encapsulated types are major (see E1 above). The impact of a separate compilation facility on language and implementation is also substantial. For example, if strong type checking is to be enforced between separately compiled modules, the compiler must ensure that symbol table information is retained along with object code. The interaction between separate compilaion and "tagged" data for debugging was cited above (D3).

3. Compile Time Scope Determination (F3).

a. Degree of compliance: T

b. TACPOL satisfies this requirement completely, via its namescope rules. It might be pointed out, however, that the wording of the behavior of the GO TO statement [3, p. 69]

could be improved, since it is possible to interpret the description in such a way that dynamic determination of the destination point is implied.

4. Libraries Available (F4).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. Serving the purpose of libraries are a variety of "intrinsic procedures" provided in the language for the performance of various utility functions (such as trigonometric and conversion routines). As mentioned in connection with F2, a compool facility is alluded to but not defined in [3]; presumably, such a feature would allow incorporation of predefined procedures, programs, or data.

c. In order for TACPOL to comply with F4, a specification of the compool facility is needed.

5. Library Contents (F5).

a. Degree of compliance: P

b. The compool facility, if specified in the language, would likely satisfy the majority of points in this requirement. However, there is no indication that routines written in other languages are to be available. Provision for such routines has minor effect on the language but presents major implementation problems because of the language-specific conventions governing parameter passage.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: P

b. Because of the lack of explanation in [3] concerning compools, it is not clear whether TACPOL satisfies this requirement. However, there is nothing in the language which would prevent TACPOL from complying, given a suitably defined compool facility.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

AD-A037 639   INTERMETRICS INC CAMBRIDGE MASS   F/G 9/2
CANDIDATE LANGUAGES EVALUATION REPORT.(U)
JAN 77 B M BROSGOL, R E HARTMAN, J R NESTOR   DAHC26-76-C-0006

UNCLASSIFIED   IR-217   USACSC-AT-76-11   NL

2 OF 6
ADA037639

b. TACPOL partially satisfies this requirement, via its file manipulation facilities; however, the absence of edited (i.e., stream-oriented) I/O is a serious drawback. In addition, the interfaces to machine-dependent capabilities other than the file system is unspecified in [3].

c. As stated in the Tinman, "there is currently little agreement on standard operating system, I/O, or file system interfaces." Because of this, it is a major undertaking to attempt to define such interfaces in a language.

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The sequential control structure is the normal flow of control. For conditional execution the IF statement and SWITCH intrinsic procedure may be used. The DO statement handles iterative execution. Facilities for recursion, parallel processing, and exception and interrupt handling are limited (see G5 through G8 below).

c. Discussions of the scope of modifications needed to bring TACPOL into compliance with G1 are presented below, where the individual sub-requirements are considered.

2. The GO TO (G2).

a. Degree of compliance: P

b. TACPOL's GoTo statement [3, p. 69], which allows a transfer of control to a point (label) in any lexically enclosing block or procedure, is more general than G2 requires. The modification needed to bring TACPOL into compliance with G2 is relatively minor, both to language and implementation. In fact, the change amounts to a simplification, since only the most local scope need be searched to locate a GoTo's destination.

3. Conditional Control (G3).

a. Degree of compliance: PT

b. TACPOL's IF statement [3, Section 10.8.5] and SWITCH intrinsic procedure [3, Section 10.11.6] satisfy this requirement to a high degree with the following exceptions.

(1) The IF statement is not "fully partitioned:" i.e., the ELSE clause is optional.

(2) The dispatch expression for the IF statement is a bitstring and not simply a Boolean expression.

(3) Lacking discriminated union, TACPOL does not permit selection based on values of such types.

c. It might also be pointed out, in light of the importance of the CASE-statement effect for conditional control, that the SWITCH procedure should be changed to become a separate statement; this is a minor revision.

d. The modifications implied by b(1) and b(2) above are minor. The implications of discriminated union were considered in connection with A7.

4. Iterative Control (G4).

a. Degree of compliance: P

b. TACPOL's DO statement [3, Section 10.8.6] meets this requirement fairly well: e.g., the control variable is local to the loop, entry is permitted only through the head of the loop, and the common special case of a fixed number of iterations is handled by a specialized control structure. A discrepancy between TACPOL and G4 is that the termination condition in the DO statement is only tested at the beginning of the loop. Also, there are a few drawbacks to TACPOL's DO statement: a control variable must always be specified, even in the presence of a WHILE clause; and the WHILE clause's termination condition is an arbitrary bitstring and not simply a Boolean expression.

c. The changes needed to bring TACPOL into compliance with G4 are not major. An EXIT statement such as the one in CS-4, a fairly simple addition to the language, would serve to allow the termination condition to be checked at arbitrary points in the loop. An iteration of the form DO n TIMES ... END would permit loops with no control variable. Restricting the WHILE clause's termination condition to a Boolean expression, useful for program reliability, is a simple modification.

5. Routines (G5).

a. Degree of compliance: P

b. The only recursion allowed in TACPOL is at program level; as stated in [3, p. 47]: "If during the execution of the body of a proper procedure, that procedure can itself be invoked, the procedure must be a program."

c. Providing recursion in TACPOL, although a fairly localized change with respect to the language, has a major impact on implementation. The issue of whether recursion should be supported in a language is one of the fundamental decisions which will shape implementation strategy (with respect to run-time storage management); it is not wise to consider recursion as an add-on feature to an implemented language. In addition, recursion in a programming language is generaly accompanied by other facilities which make comparable demands for run-time support and which share with recursion a realm of programming applications. Such facilities include pointers and run-time determinable array bounds. The provision of such facilities in TACPOL would basically entail a redesign of the language.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. TACPOL contains no facilities for creating and terminating parallel (or pseudo-parallel) processes. The only features which take concurrency of operation into account (*) are the file handling statements READ and WRITE [3, Sections 10.10.7 and 10.10.8]; when the return option is present, data transmission occurs concurrently with the execution of the statements following the READ or WRITE.

c. The addition of parallel processing is a major change to both language and implementation. Also, the conflict between TACPOL's basic goals and the Tinman's objectives is again illustrated. In order for TACPOL to comply with the intentions of G6, a safe facility for dealing with critical regions (i.e., to prevent the occurrence of race or deadlock conditions) is needed. However, the run-time overhead implied by such a facility is counter to TACPOL's efficiency objective.

7. Exception Handling (G7).

a. Degree of compliance: P

(*)The ENQUEUE and DEQUEUE statements [6, Sections 8-11 and 8-12], which are relevant to the use of shared resources, are also concerned with concurrency. These statements are not included in the primary reference for TACPOL [3], however.

b. TACPOL contains only a limited set of features to support exception handling. The TACPOL ON statement, despite using the same keyword as a fairly powerful facility in PL/I, is basically a form of conditional statement which detects if a previous file-handling command resulted in any of a fixed number of situations (e.g., end-of-file encountered, record-key-not-found). Condition declarations [3, Section 10.4.10] allow the programmer to monitor certain run-time conditions (e.g., division by zero, fixed-point overflow, usage of a programmer-named entity); however, the actions taken are system-dependent and not programmer specifiable.

c. The addition of exception handling to TACPOL is a major and difficult change, with respect to both language and implementation. The problems are, first, that there is little agreement concerning the proper exception facilities to support G7; second, that the interactions with other features (e.g., parameter types, parallel processing) are subtle; and third, even programs not using the facility must pay some costs associated with it.

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. TACPOL contains only a limited set of features in this area. The WAIT statement [3, Section 10.10.15] has the effect that "the continued execution of the program is delayed until all operations requested pertaining to the designated file (partition) have been completed." The ENQUEUE and DEQUEUE statements (see G6 above) allow synchronization via exclusive access to resources.

c. The facilites called for in G8 are refinements of the requirements of G6 and G7. The discussions there are applicable here: the modifications needed have major impact on language and implementation.

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement to a high degree. However, the similarity in syntax between CELL and GROUP declarations, and the use of the INIT keyword to denote assignment of value to constants, are in conflict with the Tinman's criteria of "clarity and readability of programs."

c. The modifications needed to bring TACPOL into compliance with H1 are minor syntactic changes.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. TACPOL satisfies this requirement completely. However, the complete absence of operator-definition facilities in TACPOL, which helps the language comply with H2, brings it into conflict with E4.

3. Source Character Set (H3).

a. Degree of compliance: T

b. TACPOL satisfies this requirement completely, containing a fifty-character alphabet [3, Section 10.3.1] which includes the 26 letters, 10 digits, and the following special characters:

+ - * / = ( ) . , ; : ' $ space

This set is a subset of 64-character ASCII. It should be noted, however, that TACPOL allows a larger set of characters in comments and non-numeric literals.

4. Identifiers and Literals (H4).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement fairly well, with the following differences:

(1) The effective break character for identifiers (the underscore) is not part of TACPOL's official

character set. There is no break character for literals.

(2) Although identifiers may have arbitrary length, only the first five and last three characters are significant with respect to distinguishing one name from another [3, Section 10.3.7].

c. The modifications needed to bring TACPOL into compliance with H4 are minor lexical changes.

5. Lexical Units and Lines (H5).

a. Degree of compliance: PU

b. This requirement consists of a main characteristic ("no continuation of lexical units across lines") and an exception (inclusion of end-of-line in literal strings). TACPOL complies with the main characteristic, but whether it satisfies the exception is implementation-dependent. Specifically, end-of-line may appear in literal strings if and only if the <OTHER MARK> character category includes the end-of-line character(s).

c. The change needed to bring TACPOL into compliance with H5 is to allow end-of-line character(s) as an alternative of the <CHAR SYMBOL> category [3, p. 35].

6. Key Words (H6).

a. Degree of compliance: P

b. TACPOL does not satisfy this requirement very well. The language contains over 100 key words [3, p. 94], all of which are reserved, including the names of all intrinsic procedures and four single characters (B, E, L, S) used in literal forms. There is no strong reason why intrinsic procedure names and the above-mentioned single characters could not be removed from the list of reserved words.

7. Comment Conventions (H7).

a. Degree of compliance: P

b. TACPOL partially meets this requirement. The comment delimiters are "/*" and "*/"; thus comments will not automatically terminate at end-of-line as required in H7.

c. The modifications needed to bring TACPOL into compliance with H7 are minor lexical changes.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. TACPOL completely satisfies this requirement, as evidenced in its syntactic rules.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement, with the exceptions that some data references are not legitimate procedure calls (viz., when dimension marks [3, Section 10.6.3] are used) and that some procedure calls are not legitimate data references (when the number of arguments exceeds the maximum number of dimensions allowable in a table).

c. The second exception cited in subparagraph b above is fairly easily remedied in the language: the maximum number of table dimensions should be relaxed (and specified in the language, in accordance with I4). The dimension mark issue is not quite so easily resolved, however, since such notation is a conventional form; its elimination would be in conflict with H1. Also, one of the aspects of TACPOL which helps the language comply with H9 -- the fact that group component names are known at the "top level" instead of being referenced via qualification forms -- violates the spirit of H1, since it prohibits the reuse of field names in different groups.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. TACPOL partially fulfills this requirement, with the following exceptions:

(1) The = symbol is used for both assignment and equality.

(2) FIXED(n) has different meanings in data declarations and file declarations.

(3) Value parameters may be assigned to [3, Section 10.4.8] but value quantities may not [3, Section 10.4.7]. (This is an inconsistency in the reference manual's terminology; the syntax uses different key words in the two cases -- VALUE vs. INIT.)

(4) RETURN has different interpretations depending on whether it appears in an I/O statement or a return statement. (The use of the keyword RETURN in a function procedure's return statement is mentioned in [6, p. 3-1]. The primary reference manual alludes to the return statement [3, Section 10.4.9a] but does not describe its syntax.)

c. The modifications needed to bring TACPOL into compliance with H10 are minor syntactic changes. For example, EQ should replace = as the relational operator (this would be consistent with the other operators [3, p. 65]).

Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: F

b. We interpret "default" here to mean "undefined result"; this appears to be the Tinman's intention, since I1 claims that the only alternative to having no defaults is "implementation dependent defaults with the translator determining the meaning of programs."

c. Under this interpretation, TACPOL fails to satisfy I1. Undefined conditions abound, from order of evaluation in expressions and initial values of variables to situations best regarded as run-time errors (e.g., fixed-point overflow and high-order truncation, subscript out of bounds, the results of intrinsic functions such as MAX and MIN). In addition, the reference manual is incomplete in describing certain subtle behavior and thus causes implementation dependencies. As an example, when one string is assigned to another, the manual does not specify whether the constituents are copied in ascending or descending order. Since TACPOL allows the SUBSTR function on the left side of assignment statements, the different orders may yield different results. Consider the string S with value 'ABCDE' and the assignment statement:

```
SUBSTR(S, 2, 3) = SUBSTR(S, 3, 3);
```

If copying is done in ascending order, the new value for S is 'ACDEE'; if descending, 'AEEEE'.

d. Substantial modifications are needed to bring TACPOL into compliance with I1. With respect to the language definition, the behavior of every undefined or erroneous program entity must be specified. The effect of such a modification on implementation is a degradation in run-time efficiency (and thus a conflict with J1) in light of the checking which is implied.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. Of the three kinds of defaults specified in I2 (data representations, open vs. closed subroutine calls, and

reentrant vs. nonreentrant code generation), TACPOL contains facilities for only the first. By specifying PACKED or ALIGNED on his data structure declaration, the programmer can override the language-defined default for the packing attribute (ALIGNED for arrays, and PACKED for groups, tables, and cells).

c. It is not a major change to introduce defaults for open vs. closed subroutine calls, and reentrant vs. nonreentrant code generation. There are, however, interactions between such facilities and the recursive routine capability called for in G5. For example, recursive routines must be compiled closed (otherwise an unbounded code expansion would result) and reentrant. The implication is that the rules for no defaults called for in I2 will have to have "special cases," to prevent the programmer from specifying open or nonreentrant on recursive routines.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. TACPOL contains no facilities for parameterizing the object machine configuration. The language is highly dependent on the particular hardware for which it was defined (the AN/GYK-12); these dependencies permeate the language. For example: the defining document specifies the essential attributes of the object computer [3, p. 27]; limits on such values as precision, scale factor, and string sizes are dictated by AN/GYK-12 characteristics; the absence of floating point from TACPOL is due to lack of hardware support.

c. It is not a simple change to the language to add the facilities called for in I3. The distinction between short and long numeric should be removed, but (as was pointed out in D4) this can have ramifications on efficiency. The problem is that if the environmental inquiry facility implicit in I3 is intended to be used for guaranteeing portability, a fair amount of language complexity ensues. For example, in some programming environments portability should be enforced (i.e., the use of non-portable constructs should be prevented), whereas in other environments efficiency constraints may dictate deviations from such conventions.

d. The impact of I3 on existing implementations will be sizable, unless these implementations were quite careful in encapsulating their assumptions concerning object machine charactersitics.

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. TACPOL has no facilities for conditional compilation. It is not a major change to add a simple capability which captures the major intentions of I4; the effects are fairly localized, with respect to both language and implementation. The difficulty lies in deciding which features to leave out, so that the added capability is in fact simple.

5. Simple Base Language (I5).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement fairly well. It is not an extensible language; thus the entire language can be regarded as the "base" or "kernel." In spite of this, TACPOL is quite simple (in fact, its simplicity is one of its strongest advantages). A minor example of a duplicated feature is INIT and = in constant declarations (this was discussed above in connection with D1 and D3).

c. No major changes are implied by I5, unless one interprets this requirement as calling for the existence of extension facilities in the language. However, it should be pointed out that the simplicity of TACPOL is at the expense of program reliability; an example is the preponderance of "undefined"s in contexts which should be program errors.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The language definition establishes limits on the number of array dimensions (three) and the effective length of identifiers (eight); however, it does not restrict the number of nested parentheses levels in expressions or the number of identifiers in programs.

c. It is a minor change to the language to specify limits as required in I6. The major problem is that of deciding what values these limits should have.

7. Object Machine Restrictions (I7).

a. Degree of compliance: P

b. The absence of floating-point from TACPOL, due to the absence of floating-point hardware on the AN/GYK-12, is a violation of I7. The implications of adding a floating-point data type to TACPOL were discussed in connection with A2 above.

Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. TACPOL satisfies this requirement to a high degree, as illustrated in the following subparagraphs.

(1) The restriction that string, array, and table bounds be known at compile-time avoids run-time overhead for "dope vectors."

(2) The requirement that the "length" argument to the SUBSTR function be known at compile-time avoids run-time overhead during string assignment.

(3) The cell feature, which realizes free union as opposed to discriminated union, avoids the latter's run-time "tag" checking.

(4) The parameter passing mechanism is fairly efficient. Since only scalars may be passed by VALUE, there is no implicit copying of large tables or groups; quantity parameters can be implemented efficiently via index registers.

(5) The data representations in TACPOL are keyed to the hardware characteristics of the AN/GYK-12; e.g., bitstrings are limited to 32 bits (the word length), fixed-point precision is limited to 31 (for short numeric data) and 62 (for long numeric data).

(6) The (only) justification for the multitude of "undefineds" in the TACPOL specification is one of efficiency; there is no automatically-provided run-time code to check for such conditions as high-order truncation or subscript out-of-bounds.

(7) In short, there is no reason that a good TACPOL compiler should not generate highly efficient target programs.

c. In satisfying J1, however, TACPOL comes into conflict

with a variety of requirements oriented around language power and program reliability (e.g., A6, A7, B5, D6, G5). There are no simple solutions to this conflict.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: PU

b. Since TACPOL does not specify order of evaluation for expression constituents, optimization may change program effect. For example, consider the expression F(1)+F(2) where F assigns the argument to a global variable X and then returns this as its value. Depending on which term of the sum is evaluated first, X will have the final value 1 or 2.

c. Although this requirement pertains more to the translator than to the language, TACPOL could facilitate compliance with J2 by specifying order of evaluation for side effects within an expression. As noted above in connection with C1, however, this has potential conflicts with efficiency.

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. TACPOL partially satisfies this requirement. The CODE statement [3, Section 10.8.8] permits the programmer to descend into MOL, but allows this at arbitrary places in the program. (The Tinman requires that MOL insertions be limited to the bodies of compile-time conditional statements, which are absent from TACPOL.)

c. The effort in encapsulating machine language insertions as specified in J3 comes mainly from the addition of a conditional compilation facility; see I4 above.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. TACPOL satisfies this requirement only in a limited fashion, through group, table, or cell definitions with packing attributes specified. It is not possible to obtain arbitrary storage layouts via these facilities, nor can the programmer define "don't care" fields or associate identifiers with machine addresses.

c. The addition of a facility for specifying the detailed machine-dependent representation of data objects is a localized change with respect to language and implementation. However, the inclusion of run-time determinable array bounds (as required in A6) and discriminated unions (A7) complicates such a facility because the programmer will have to be aware of how dope vectors and tag fields are represented by the compiler.

d. To allow the user to specify time/space tradeoffs to the translator, TACPOL could incorporate optimization directives as found, e.g., in JOVIAL. This is not a difficult addition to the language. The effect on implementation can be substantial depending on the degree of optimization which is intended by the directives. There is also the possibility for conflict between a directive and another language feature (e.g., providing a time-optimization directive for a program which includes packed records).

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: F

b. TACPOL does not fulfill this requirement; the programmer has no control over whether a routine is compiled in closed or open form.

c. It is a minor change to allow an attribute to be specified in a procedure declaration which will indicate open vs. closed compilation. The default should be closed compilation.

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: P

b. Several facilities provided by TACPOL imply the existence of run-time support which is generally provided by an operating system. Condition declarations [3, p. 49] require that zero-divide and fixed-overflow exceptions be trapped, so that a "snap" procedure may be invoked. The file processing features [3, pp. 73 ff] require supervisory routines to ensure efficient and secure use of resources.

c. Bringing TACPOL into compliance with K1 would involve removing condition declarations and file processing from the language. However, that would be in conflict with other Tinman characteristics (B10, G7).

2. Program Assembly (K2).

a. Degree of compliance: U

b. The Compool facility would be useful for supporting this requirement; however, this feature is not documented in [3]. The fact that a program must be a procedure [3, p. 73] implies that some kind of separate compilation facility was likely envisioned for TACPOL, but this is not described in [3].

c. The changes needed to bring TACPOL into compliance with K2 are not major from a language viewpoint, since their effects are relatively localized and they need not be complex. However, as noted in the Tinman, implementation impact may be sustantial (e.g., enforcing type checking across the interface between separately compiled programs).

3. Software Development Tools (K3).

a. Degree of compliance: U

b. This requirement is not addressed in [3], except for a small set of debug facilities realized in condition declarations. The provision of tools such as the ones described in K3 has a major impact on language and implementation, since language/tool environment must be designed as an integrated system.

4. Translator Options (K4).

a. Degree of compliance: U

b. The options described in K4 are not addressed in TACPOL's defining document [3]. Their incorporation would not have large impact on the language, but could have major effect on implementation; a cross-reference facility is not necessarily a simple addition to a compiler.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: F

b. TACPOL contains none of the facilities required in K5. It is a major change, to language and implementation, to provide such features, especially since there is no widespread agreement on the kinds of specifications which are both practical and powerful enough.

Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

2. No Subset Implementations (L2).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

3. Low-Cost Translation (L3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition. However, the simplicity of TACPOL makes it reasonable to expect implementations to achieve the objectives of L3.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition. However, the orientation of TACPOL to the charactristics of the AN/GYK-12 makes it unlikely for efficient object code to be produceable for a variety of target machines.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. The language definition is silent concerning whether and when the checking required in L6 is carried out. As stated in [3, p. 31]:

> Whenever a semantic definition uses the terms "such as," "must be," "may only be," "cannot be," or "is undefined," the intent is to delineate proper syntacitc constructions which have meaning in TACPOL and those which have no meaning in TACPOL. In the latter case, meaning may be attributed to those constructions elsewhere, but is not here.

7. Diagnostic Messages (L7).

a. Degree of compliance: U

b. There is no discussion of diagnostic error or warning messages in the language definition. The passage cited in connection with L6 above implies that the detection of illegal constructs is implementation dependent.

8. Translator Internal Structure (L8).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition. It may be noted that TACPOL has sufficient facilities to enable a compiler to be written in the language, but that the absence of recursion and dynamic allocation (e.g., for arrays) and the relative lack of type checking are drawbacks.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: P.

b. As an existing language, TACPOL is composed from features which are within the state of the art. The problem is that advances in the state of the art have revealed problems (with respect to program reliability) in facilities such as those incorporated into TACPOL which were previously touted for efficiency and notational freedom. It is not realistic to expect to meet the Tinman requirement that "any design or redesign [to TACPOL] which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project."

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. The defining document for TACPOL partially satisfies the requirements of M2. Although the semantics are not specified formally (as in the Vienna definition language), the execution model presented in [3], in which a language feature is explained in terms of a "rewriting rule," provides a clean and clear description. Examples appear in the explanation of the cell definition [3, pp. 43-44], the value declaration [3, p. 44], the procedure invocation [3, pp. 45 ff], and the do-statement [3, p. 71].

c. Problems with the document are: occasional incompleteness (e.g., omission of descriptions for COMPOOL and the ENQUEUE and DEQUEUE statements) and ambiguity (see the example under F3 above); lack of examples; unconventional organization of material; lack of an index and glossary of terms; lack of a summary of the syntax rules.

3. Language Documentation Required (M3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition. Comments on the language definition document are given in connection with M2 above.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

Section XV. CONCLUSIONS REGARDING TACPOL

1. Conflicts between Objectives of TACPOL and Tinman.

a. One immediate conclusion of the evaluation presented in this chapter is that there are sizable (and, for practical purposes, insuperable) differences between TACPOL and the Tinman characteristics. The cause of these disagreements lies in the differences between the priorities of the objectives sought by TACPOL and the Tinman. For TACPOL, the primary goals were object-code efficiency and language simplicity. Portability was never a serious concern, since the language was designed to be used on (and thus to exploit the characteristics of) a particular hardware configuration (the AN/GYK-12). Although software reliability and maintainability were objectives for TACPOL, at the time of the language's design the knowledge concerning the impact of language features on programming methodology and reliability was not nearly so developed as it is today. And in view of the limited range of applications envisioned for TACPOL (tactical software, as opposed to compiler writing, for example) language power was regarded as a goal which could be compromised in the interests of attaining efficiency and simplicity (thus the prohibition of dynamic array bounds).

b. The Tinman, on the other hand, placed an entirely different set of priorities on the various language goals (assuming that the amount of attention paid in the Tinman to characteristics which support an objective reflects the priority of that objective). Program reliability and maintainability and language power are the primary objectives which a Tinman-like language would seek to attain. If the efficiency sought in the Tinman's section J is to be realized also, a heavy price in language complexity appears inevitable.

2. Summary of Major Areas of Conflict between TACPOL and Tinman.

a. Data and Types. There are several serious conflicts between TACPOL and the Tinman in this area. This is illustrated by a variety of omissions from TACPOL: there is no type checking for quantity (i.e., "by reference") parameter passing; there is no data-type for floating point arithmetic; there is no discriminated union facility. In

addition, TACPOL's requirement that array bounds be known at compile-time is in conflict with the Tinman.

b. Operations. The absence from TACPOL of formatted I/O and exception signalling on truncation, and the presence of implicit type conversions, are in conflict with the Tinman.

c. Expressions and Parameters. The lack of compile-time evaluation of constant expressions and the limited parameter passing mechanism in TACPOL are in conflict with the Tinman.

d. Variables, Literals and Constants. Deficiencies in TACPOL in this area are the absence of provisions for declaration-associated initialization of variables, the lack of a pointer facility, and the limitations on data structuring.

e. Definition Facilities. This is the area in which TACPOL fares most poorly. There are no facilities in TACPOL for encapsulating type definitions in the sense required by the Tinman. (In fact, there are no facilities for defining data types at all in TACPOL.) There is no provision in TACPOL for operator extension. TACPOL does not allow data definition via enumeration of literal names.

f. Scopes and Libraries. TACPOL has no major conflicts with these requirements, except that the semantics of the Compool construct are not explicitly specified in the language definition [3].

g. Control Structures. Conflicts with the Tinman in this area stem from TACPOL's lack of recursive routines and the absence of parallel processing and exception handling.

h. Syntax and Comment Conventions. There are no major conflicts between TACPOL and the Tinman in this area.

i. Defaults, Conditional Compilation and Language Restrictions. Conflicting seriously with the Tinman is the preponderance in TACPOL of "undefined" conditions which should be detected as errors. In addition, TACPOL does not parameterize the object machine, nor does it provide a facility for conditional compilation.

j. Efficient Object Representations and Machine Dependencies. A conflict in this area is that, since TACPOL

leaves the order of evaluation unspecified for expression constituents, optimization may change program effect.

k. Program Environment. TACPOL lacks a variety of facilities, in connection with software development tools, translator options, and assertion-like specifications, which are required by the Tinman.

l. Translators. The main conflict with the Tinman in this area is that the orientation of TACPOL to the characteristics of the AN/GYK-12 makes it unlikely for efficient object code to be produceable for a variety of target machines.

m. Language Definition, Standards and Control. These characteristics do not pertain to the language per se; thus it is not applicable to speak of conflicts between TACPOL and the Tinman in this area.

3. Unnecessary Features in TACPOL.

One of the objectives of this report is to identify language features which are not needed to satisfy (but do not conflict with) the requirements, with a recommendation as to whether such features should be kept or possibly eliminated. In the case of TACPOL, there are no features which fall directly into this category; however, the question of whether a feature is or is not "necessary" to satisfy a requirement is somewhat subjective.

4. Recommendations concerning TACPOL.

On the basis of the evaluation conducted in this chapter, we conclude that an attempt to modify TACPOL to bring it into compliance with the Tinman would be inadvisable. The many fundamental differences between the two, caused by the basically different objectives desired, would imply a substantial redesign effort and a new implementation. It is not realistic to expect the "flavor" of TACPOL to be retained in such a new language. Thus, the advantages typically cited for choosing an existing language (availability of implementations, documentation, trained programmers) would effectively be forfeited. In summary, TACPOL in its present form is not suitable with respect to satisfying the Tinman requirements, and no language which differs from TACPOL in only minor ways will be substantially better.

CHAPTER 4

CS-4 EVALUATION

Section I. LANGUAGE SUMMARY

1. Lexical Properties.

CS-4 is a free-format language which uses a subset of the 128 character ASCII character set. The subset consists of the 94 printing characters, the carriage-return and line-feed characters, and the space (blank). The space character is significant in separating tokens. Some keywords are reserved and some are reserved in specific contexts. Identifiers can consist of letters, digits and underscores, but must begin with a letter, end with a letter or digit, and not exceed 32 characters in length. There are two comment forms. The first is determined by "%" and a new-line sequence and can contain any characters. The second is delimited by "{" and "}" and can contain lexical tokens.

2. Data Types.

a. Built-in Types. The data types (called modes in CS-4) which are built into the language are displayed in Figure 3. The BOOLEAN mode is used for representing logical truth values. The STATUS mode is used for representing ordered sets of symbolically named data. The INTEGER mode is used for representing data whose behavior is to be that of the set of mathematical integer values. The REAL mode is used for representing numeric data whose values may be inexact and which fall in a range wider than that supplied by the INTEGER mode. The FRACTION mode is similar to REAL except that values must lie between -1.Ø and 1.Ø. The ARRAY mode is used for representing aggregate data whose components have identical traits. The STRING mode is used for representing data comprising sequences of ASCII characters. The SET mode is used for representing the powerset of the enumerated values of an underlying STATUS mode. The COMPLEX mode is used for representing complex numbers. The VECTOR mode is used to represent all one-dimensional sequences of REAL values. The MATRIX mode is used to represent the set of real matrices. The

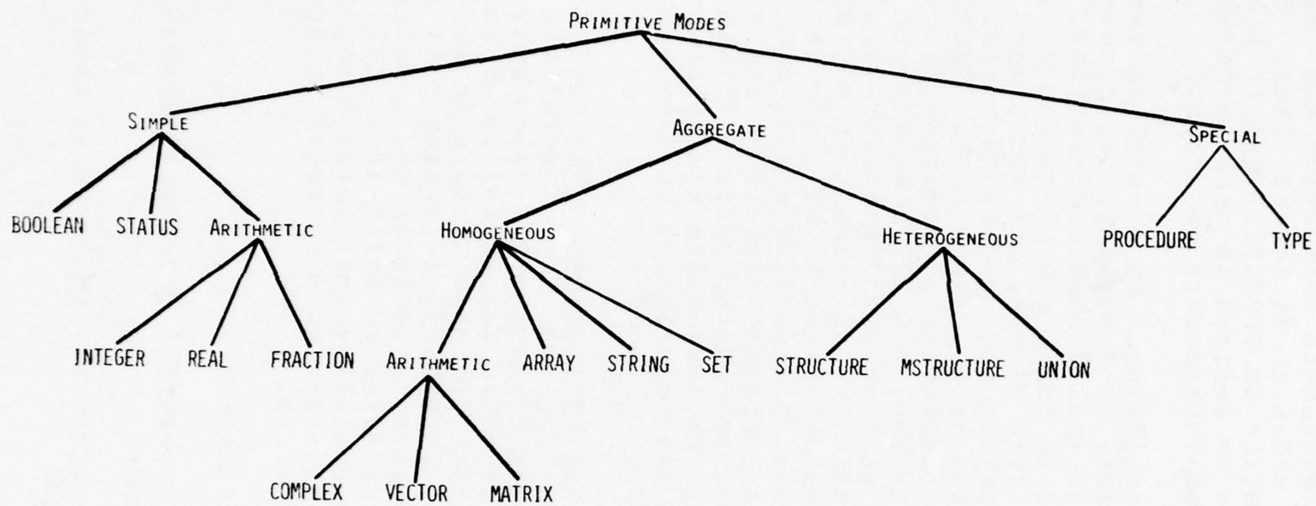

Figure 3. The Primitive Modes in CS-4

STRUCTURE mode is used for representing aggregate data whose components may be of different modes. The UNION mode is used for representing aggregate data which can take on values from different modes at different times. The MSTRUCTURE mode is similar to STRUCTURE but it allows the data representation to be defined in machine-dependent terms and/or allocated at absolute memory locations.

b. Definitional Facilities. As a complementary capability to procedures, CS-4 supports the concept of data abstraction. These are called MODE declarations. A MODE declaration is similar in form to a procedure definition. It defines a new data type (mode), and associates a name with it. MODE declarations may specify parameters required when the MODE is invoked (e.g., RANGE on the INTEGER mode). Users may specify the operations of assignment, comparison, initialization, termination, and I/O routines for defined modes. The lasnguage will supply "standard" versions of most of these if the user wishes. The user may also specify any other operations which are to be part of the interface of the mode. Parameters of type TYPE may be specified in mode declarations (e.g., element-type in an ARRAY mode). Users may specify which procedures defined within mode-definitions are to be accessible and which are to be "hidden"; this is the same capability as used with separate compilations and ACCESS.

c. Type Checking and Conversions. The CS-4 language strongly enforces data-type checking. Operations in the language accept operands of designated types only; mismatch of types is normally detected and signalled at compile-time. When a user must postpone certain decisions until run-time (for example, array size determination) the language must correspondingly postpone the completion of its checking also. CS-4 provides no built-in implicit data type conversions, and does not allow user-defined implicit conversions. Explicitly-invokable conversions are provided, and may be defined by users.

3. Procedures and Parameters.

a. CS-4 provides a powerful procedure-capability with full interface checking.

b. Procedure parameters may be bound to their arguments by COPY, REF, and NAME.

c. For full disclosure of programmer intent, for checking, and for efficiency of generated code, the parameter binding specification must designate whether the procedure wishes to obtain the value of the argument, to modify the argument, or both.

d. Keyword parameters are supported.

e. Procedure code can be expanded in line at each call (OPEN), or can be conventionally compiled (CLOSED).

f. Procedures can be passed as arguments in procedure calls.

g. Recursion is not permitted.

h. The MPROCEDURE procedure type allows definition of procedure code in target-machine assembly language. The high-level interface definition and interface checking are preserved.

i. The EPROCEDURE procedure type is a template which describes an "external" procedure written in a language other than CS-4.

4. Statements.

a. Assignment Statement. The action of an assignment statement is to invoke the special ASSIGN routine associated with the mode of the designated object.

b. If Statement. An If Statement indicates a segment of program text whose execution depends upon the run-time state of a BOOLEAN value. The ELSE phrase is optional.

c. Case Statement. A Case Statement permits the dynamic selection of one list of statements to be executed from a group of such lists. An OTHERWISE phrase is optional.

d. Repeat Statement. A Repeat Statement permits the repeated execution of a list of statements. Optional FOR and WHILE phrases are permitted to control the repetition.

e. Exit Statement. An Exit Statement is used to immediately terminate the execution of a labelled block or repeat statement.

f. <u>Procedure Statement</u>. A Procedure Statement is used to invoke a group of statements declared as a procedure.

g. <u>Return Statement</u>. A Return Statement is used to terminate execution of a procedure and return control to the point immediately following the procedure's invocation.

h. <u>Go To Statement</u>. A Go To Statement is used to transfer control to a labelled statement; restrictions on its use prevent such error-prone situations as jumping into a repeat-loop or out of a procedure.

i. <u>Signal Statement</u>. A Signal Statement is used to invoke a procedure which has been declared for exception handling.

j. <u>Resignal Statement</u>. A Resignal Statement is used within a signal-handling procedure to generate the same signal which caused control to pass to the current handler. The newly-generated signal will be processed by a handler which precedes the current handler on the dynamic control flow chain.

k. <u>Abort Statement</u>. The Abort Statement is a less restricted form of Go To Statement which may only appear in the body of signal-handling procedures and which can transfer control to a label in the immediately containing program or procedure.

l. <u>Begin Block</u>. A Begin Block is used to group a sequence of declarative and executable statements together for treatment as a single compound statement.

m. <u>Update Block</u>. An Update Block is used to supervise the access to data which are shared by separately-executing processes.

5. Storage Allocation.

a. The storage class of an object designates the times of allocation and deallocation of the object. The scope of reference to an object never exceeds the scope of its allocation.

b. AUTOMATIC objects are allocated at block-entry and deallocated at exit.

c. STATIC objects are allocated for the lifetime of the process in which they are declared.

d. SHARED objects are allocated at the first initiation of a process in which they are declared, and deallocated at the termination of the last such process. All processes share the same object if its storage class is SHARED.

e. There are two sub-classes of sharing:

(1) SHARED(PROTECTED) objects have compiler-generated code which guarantees orderly and deadlock-free access by concurrent processes.

(2) SHARED(UNPROTECTED) objects must be protected explicitly by user code.

f. Absolute memory-location assignment is also provided.

6. Process Scheduling.

The CS-4 Operating System Interface contains facilities for creating processes and specifying priority levels or real-time constraints. In addition capabilities are provided for process-state inquiry, process interruption and resumption, process termination, shared data, and process synchronization.

7. Files and I/O.

The CS-4 Operating System Interface supports a file system. This system contains a hierarchically organized directory. Files may be stream-oriented or record-oriented and may have one of the following three organizations: consecutive, random-access, indexed sequential. In addition, CS-4 supports both edited and unedited I/O.

8. Exception Handling.

a. Comprehensive error detection is provided at compile time, program binding time, and run time.

b. A software-invokable signal mechanism, which is oriented to the dynamic procedure-calling sequence, is provided to complement the ordinary static mechanism.

c. Procedures can be designated as signal handlers. Signals can pass data to parameters of signal handlers. Type checking is performed on these parameters.

9. Compile-Time Facilities.

a. Compile-Time Evaluable Expressions. Many of the built-in operators and functions will be invoked at compile-time provided all of their arguments are known at compile-time. Such expressions can be used in contexts requiring compile-time known values.

b. Modular Composition Capability. The structuring properties of CS-4 are intended to support large-scale programming efforts involving many programmers. The unit of compilation is a PROGRAM. The interface (exportable capabilities, data objects and types, etc.) of each PROGRAM is explicit. All of the components of the interface of one PROGRAM may be accessed in another PROGRAM by means of an ACCESS statement which names the desired PROGRAM. Unlike a conventional INCLUDE directive, which fetches source text whose relationship with external procedures (etc.) cannot be checked, the CS-4 ACCESS statement causes direct examination of the compiled program itself. Thus, the checking across separate compilations is as thorough as that within a single compilation. Subsets of program interfaces may be ACCESSED as well as the entire interface. Renaming may be specified on the using side, to eliminate name conflicts. PROGRAMS may export selected capabilities which are defined internally or accessed. Thus, the program composition process can be repeatedly applied as needed to construct large-scale program structures from tractable components in a modular fashion.

c. Administrative Language Restrictability. CS-4 restricts the use of certain language capabilities. Each program must contain a declaration of any potentially dangerous features it uses. The compiler will flag any violations of these restrictions: Go To statements, Signals, Abort statements, Disabling of checking, Mstructures, Absolute allocation, Mprocedures, Eprocedures, and Non-portability.

d. Checking Directives. CS-4 normally performs a comprehensive group of checks to detect program errors. When necessary for run-time efficiency, these checks may be

selectively disabled by compile-time directive, thereby removing run-time overhead in object code size and execution time in exchange for increased vulnerability to error.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement. When an object is declared in CS-4, the data type information that appears is called a <u>mode-invocation</u>. The <u>mode</u> of the object is simply the name that is invoked. The formal parameters of the invocation are called the <u>traits</u> of the mode. The set of values for the compile-time traits of the mode is referred to as the <u>type</u> that results from the mode-invocation. The template for the value of the run-time traits is called the <u>run-time profile</u>. Thus, both the mode and type of each object are known at compile-time. This is even true (as required by A1) for components of UNION objects, since the language always requires the assertion of which alternative of the UNION is valid.

2. Data Types (A2).

a. Degree of compliance: P

b. CS-4 satisfies this requirement almost completely. The only exception is that the language does not provide a general fixed-point facility -- the FRACTION mode (CS-4's version of fixed-point) supplies values only between -1.0 and 1.0. In addition, (although this is a fairly minor difference) there is no explicit data type for <u>characters</u>: a character item in CS-4 is of STRING mode with length 1.

c. The scope of modifications is discussed in paragraphs 4 (Fixed-Point Numbers) and 5 (Character Data) below.

3. Precision (A3).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement. Points of agreement are:

(1) Precision is a <u>trait</u> of the REAL mode; thus, real values with different precisions still have the same mode.

(2) CS-4 does permit precision specification for individual variables (In fact, the language requires such specification.)

The main disagreement between CS-4 and A3 is that the language does not allow precision specificatrions to be associated with a namescope.

c. The modifications required to the language are relatively minor. A global precision declaration would have to be added to the language and the precision specification currently required for individual variables would have to be made optional.

d. The impact on implementation would be relatively minor and would require changes only to the declaration processor which enters trait information in the symbol table.

4. Fixed-Point Numbers (A4).

a. Degree of compliance: P

b. The FRACTION mode in CS-4 partially fulfills this requirement. The points of difference (both of which are major) are that FRACTION values are limited to the interval from -1.0 to 1.0, and that there are no FRACTION literals. FRACTION literals can, however, be constructed from REAL literals, but need not be represented exactly.

c. The modifications required are fairly substantial. FRACTION mode was intended as a way of making REALs more efficient for the case when the exponent was zero but with similar semantics to the REAL mode. To conform to the fixed point requirement, FRACTION mode would have to be changed to define exact quantities with a range and step size specifiable by the user. This change would also impact the arithmetic operations that operate on FRACTION mode. Despite the substantial nature of the change, it can be accomplished relatively smoothly without any major impact on other aspects of the language.

d. The impact on the implementation would be substantial, affecting all phases of compilation. Assuming a well-structured compiler, however, the changes should be fairly localized within each phase, and could be made quite cleanly.

5. Character Data (A5).

a. Degree of compliance: P

b. CS-4 is defined in terms of a subset of the 128 character ASCII Standard Character Set. There is no provision in the language for the user to specify another character set. CS-4 provides the STRING mode as primitive but offers no explicit data type for characters. Although individual characters can be represented as STRINGs of length 1, this is not treated as any other enumeration type, since the only comparison operations defined for strings are equal and not equal. In order to compare strings for less than, or greater than, the built-in ASCII function would have to be used to convert the characters to their integer ASCII representation.

c. To modify CS-4 strings of length one (using the ASCII character set) to act as an enumeration type would be relatively minor. It would involve defining the less than and greater than operators for strings as well as defining the SET mode to operate on the set of ASCII characters. To allow the user to define his own character set and also allow literals composed of strings of these characters is not within the state-of-the-art of language implementation and is also not desirable as it implies that the lexical nature of the language can change.

6. Arrays (A6).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement fairly completely. The only disagreement is that CS-4 permits both bounds of an array dimension specification to be run-time determined, while the Tinman requires the lower bound to be known at compile-time.

c. The change to the language to require the lower bound to be known at compile-time is of an extremely minor nature and is, in fact, a simplification.

d. The implementation would require a very minor change to the declaration processor.

7. Records (A7).

a. Degree of compliance: PT

b. The UNION mode in CS-4 almost satisfies this requirement completely. However, CS-4 UNIONs require a tag to be present for reliability. CS-4's MSTRUCTUREs also satisfy this requirement to a large extent, but since a tag field is not required, complete type checking cannot be ensured.

c. CS-4 cannot be modified to meet this requirement completely because the requirement itself is inconsistent.

Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. As described in [4, Section 6.1.3], when defining a new mode, the programmer has the option either (1) to make no assignment procedure available to users of the mode, (2) to make available a routine specifically coded to capture the meaning of assignment for the new mode, or (3) to make available the standard compiler-supplied assignment routine. The points of difference between CS-4 and B1 are relatively minor:

(1) Assignment is not provided from a UNION component to the UNION mode and vice versa.
(2) The Tinman wants the user to "be able to declare variables for all data types." This is essentially true in CS-4; however, CS-4 has extended its concept of "mode" to include TYPE and PROCEDURE-valued parameters.

c. The modifications required for UNIONs are relatively minor. The assignment operator would have to be changed to permit an assignment from a component type to a UNION containing that type which includes changing the TAG. To permit assignment from a UNION type to a component type would require a run-time check of the TAG field for type compatibility.

d. The effect on the implementation would be relatively minor and localized to portions of the compiler dealing with UNION assignment.

2. Equivalence (B2).

a. Degree of compliance: P

b. The "=" operation in CS-4, which invokes the COMPARE routine [4, p. 164], satisfies this requirement fairly well. One exception, a minor one, is that while B2 requires the identity operation for floating-point values to check for identity only within the specified precision, the COMPARE routine for REALs in CS-4 tests for exact equality [4, p.

50]. However, CS-4 does provide a pre-defined procedure, REAL_EQ, that supplies the behavior required in B2. In addition, CS-4 does not allow "=" to compare objects of different types; such an attempt would be treated as a compile-time error.

c. One modification required is to make the REAL_EQ procedure the COMPARE procedure for REALs. For consistency, <, <=, >, >=, ¬= should also be changed to invoke REAL_LT, REAL_LE, REAL_GT, REAL_GE, and REAL_NE.

d. Another modification required is to change "=" to give a FALSE value when the object types are different.

e. The impact on the implementation is fairly minor.

3. Relationals (B3).

a. Degree of compliance: PT

b. CS-4 partially satisfies this requirement. The six relational operations are always defined for types resulting from invocations of numeric (INTEGER, REAL, FRACTION) and enumeration (STATUS) modes. However, there is no way of inhibiting these operations when an ordering is not desired.

c. The language modifications required to inhibit the relationals when an ordering is not desired are non-trivial. A new, unordered STATUS mode would have to be added to the language (e.g., USTATUS). Literals for this mode would have to be differentiated from literals of the ordered STATUS mode to prevent an undesirable interaction of the two modes.

d. The effect on the implementation would also be non-trivial. Although USTATUS is similar to STATUS, extra checking is required to determine equivalence of types. In the following example

```
VARIABLE X IS USTATUS ("A", "B", "C"),
         Y IS USTATUS ("B", "C", "A"),
         Z IS USTATUS ("A", "C", "B");
```

the compiler would have to determine that X, Y, and Z all have the same type.

4. Arithmetic Operations (B4).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely [4, Section 3.2.1 through 3.2.3].

5. Truncation and Rounding (B5).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely [4, Section 3.2.4].

6. Boolean Operations (B6).

a. Degree of compliance: T

b. CS-4 provides not only the operations listed in B6 (AND, OR, NOT, NOR) but also NAND, EQV, XOR, and IMP [4, Section 3.1.4.1]. Short-circuiting is supplied for AND and OR, as well as NAND and NOR.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. CS-4 does not provide the general facility required in B7. CS-4's definition of "conformable" arrays is much stricter than the definition in B7. For two arrays to "conform" in CS-4, they must have the same number of dimensions, and the same number of elements in each dimension, not merely the same number of elements as required in B7. CS-4 permits assignment and comparison between conformable arrays, but none of the other operations. Only assignment is permitted between a scalar and an array and then only in certain contexts such as array initialization. CS-4 is overly restrictive on assignments between records (STRUCTUREs) and requires the field names to be identical. CS-4 does not enforce the distinction, as required in B7, between operations defined for arrays and operations defined for modes whose object-representations are arrays. It should be noted that CS-4 supplies VECTOR and MATRIX as built-in modes that include the behavior required in B7.

c. The modifications required to the language would be fairly minor. Relaxing the restriction on STRUCTURE conformability is straightforward. Relaxing the restriction on array conformability, although simple, is undesirable as it would defeat the strong typing philosophy of the language. Extending the definition of scalar operations to conformable arrays and arrays with scalars is straightforward.

d. The effect on the implementation is minor. Relaxing the restriction on STRUCTUREs would only require the deletion of field name checking; the logical structure is already checked. Extending the operation definitions is a straightforward task.

8. Type Conversion (B8).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely with the following differences:

(1) CS-4 permits a match between an argument and a UNION formal parameter only when the types are identical. The Tinman requirement for a match when the argument mode is a constituent of the UNION mode is not met.

(2) Lacking a general fixed-point facility, CS-4 does not provide explicit conversions between fixed-point scale factors.

c. It should be noted that although CS-4 permits mixed mode arithmetic, this need not be viewed as involving an implicit conversion, but rather as instances of generics that accept arguments of different modes.

d. The modifications required to add fixed point have been discussed in connection with A4. To modify the language to permit a match when an argument is a constituent mode of a UNION mode formal parameter presents serious implementation problems when the parameter is passed by reference.

9. Changes in Numeric Ranges (B9).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. The RANGE trait does not distinguish the mode in CS-4; when an out-of-range assignment is attempted, the X_INTEGER_RANGE or X_REAL_RANGE error is signalled [4, p. 39, p. 50].

10. I/O Operations (B10).

a. Degree of compliance: PT

b. This requirement is satisfied almost completely by CS-4's Operating System Interface [5]. The only capability not provided by CS-4 is that of allowing dynamic assignment and reassignment of I/O devices.

c. Changing the CS-4 Operating System Interface to allow dynamic assignment and reassignment of I/O devices is a minor task. The main problem, however, is that such a capability may be specifically prohibited by a particular operating system.

11. Power Set Operations (B11).

a. Degree of compliance: T

b. The SET mode in CS-4 [4, pp. 253ff], which takes a STATUS type as argument, satisfies B11 completely. However, element predicate has been implemented as the selector, which may not be a good choice since S("element"):=TRUE is not as readable as INSERT(S, "element").

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

   a. Degree of compliance: T

   b. CS-4 completely satisfies this requirement. As stated in [4, p. 28]: "the order of evaluation of the operands within a given expression is effectively as if an operator's left operand is always evaluated before its right operand". It should be noted that this rule enables the compiler to modify the order of evaluation (for optimization purposes) when it can ensure that no side effects are involved.

2. Operand Structure (C2).

   a. Degree of compliance: T

   b. CS-4 totally satisfies this requirement. The only qualification is that CS-4 contains a large number of operators, and the precedence of some of these is not widely agreed upon (e.g., OUTER and CROSS for VECTORS).

3. Expressions Permitted (C3).

   a. Degree of compliance: T

   b. CS-4 satisfies this requirement completely.

4. Constant Expressions (C4).

   a. Degree of compliance: P

   b. CS-4 satisfies this requirement to a large extent [4, p. 173]. However, not all the built-in functions are compile-time invokable.

   c. The required modifications are relatively minor. Built-in functions such as ABS and SGN that are not currently compile-time invokable but which could meaningfully be used in a constant expression would have to be made compile-time invokable.

   d. The impact on the implementation would be relatively minor.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. CS-4 does not satisfy this requirement very well. Although there is a consistent set of rules for parameters applicable to procedures, exception handlers, parallel processes, and built-in operators, there is a different set of rules for parameters to user defined modes.

c. The modifications required to meet this requirement would be substantial. Mode definitions are of such a different nature than procedures that it is not clear if the rules can be made meaningfully consistent. One way this requirement can be met is to eliminate parameters to user defined modes, although this would limit the power of the language.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely, as explained in the rules for COPY, REF, and NAME binding [4, pp. 144-145, p. 149]. The only disagreement (also noted earlier in connection with B1 and B8) is that, in CS-4, formal parameters of a union type are not considered conformable to actual parameters of component types.

c. As discussed under B8, allowing actual parameters of a component type to conform to a formal parameter of a union type presents serious implementation problems in the case of "by reference" parameter passage.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. The Tinman states that "there will be only four classes of formal parameters", including one for responses to exception conditions and one for procedure-valued parameters. CS-4 has three classes of bindings (COPY, REF, and NAME), each of which may specify that the information flow is either "in", "out", or "in-out". COPYI and REFIO nearly correspond to the first two classes of parameters required in C7 except that COPYI parameters may be assigned

to locally. CS-4 has procedure-valued parameters which must have a binding of NAMEI. CS-4 provides an exception handling control facility through a signal handler mechanism, an alternative approach to exception handling control parameters.

c. In terms of procedures, CS-4 can be easily modified to eliminate all binding classes except COPYI, REFIO and a special class for procedure-valued parameters. However, as discussed under C5, CS-4 has special rules for parameters to mode definitions (e.g., they must all be NAMEI), so making the above changes would interfere with mode definitions. Changing the exception handling mechanism to conform with this requirement would be a major change and would result in added complexity.

d. Eliminating the extra binding classes for parameters to procedures would have a relatively small impact on the implementation. However, changing the parameter classes for mode definitions and changing the exception handling mechanism would be a major change to the implementation.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: P

b. CS-4 partially meets this requirement, via the MOPEN compilation class for procedures. However, although the traits of the formal parameter's mode may be omitted, the mode name itself must be specified [4, pp. 148-149].

c. It is difficult to discuss modifications to meet this requirement because of inconsistencies within this requirement as described in Appendix III. In particular the macro capability implied by the phrase "instantiated at compile-time by the characteristics of their actual parameters" does not provide the claimed "generic" capability.

9. Variable Number of Parameters (C9).

a. Degree of compliance: F

b. Since user-defined procedures always contain a fixed number of parameters, CS-4 fails to satisfy this requirement.

c. The modifications required to support this requirement would be non-trivial. The parameter passing mechanism would have to be changed to meet the requirement, possibly by augmenting it with an implicit conversion from the variable size list of parameters on the calling side to a variable size array that the procedure itself would receive.

Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely with its CONSTANT declarations [4, pp. 15-16]. It should be noted that CONSTANTs are run-time, not compile-time, evaluated.

2. Numeric Literals (D2).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. The syntax and semantics for literals are explained in [4, Sections 1.3.2 (general), 3.2.1.1 (INTEGER literals), 3.2.2.1 (REAL literals), 3.3.1 (STATUS), 3.6.1 (STRING)]. The only disagreement between CS-4 and D2 is that CS-4 lacks a literal form for one of the atomic modes (viz., FRACTION).

c. The modifications required to add FRACTION literals are fairly straightforward, dependent, of course, on the extension of FRACTIONs to general fixed-point quantities (see A4).

3. Initial Values of Variables (D3).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement. In CS-4, it is necessary for the programmer to supply an initialization specification with each variable or constant declaration. However, the special NOVALUE form [4, p. 18] may be used to inhibit initialization, and (at user option) detection of erroneous usage of variables declared with NOVALUE initialization will be performed. CS-4 also meets the Tinman requirement that there be no default initial values.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. Not having true fixed-point variables, CS-4 cannot completely satisfy this requirement. However, the INTEGER

and REAL modes have a RANGE trait used to denote numeric intervals, and the REAL and FRACTION modes have a PRECISION trait, which determines step size. Being traits of these modes, the RANGE and PRECISION traits do not result in new modes.

c. Modifications to add fixed-point to CS-4 have been discussed in A4.

5. Variable Types (D5).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. The only difference is that CS-4 does not allow the definition of subsequences of STATUS (enumeration) types.

c. The modifications to permit subsequences of STATUS would be substantial and counter to the philosophy of STATUS. CS-4 permits STATUS literals to occur in any number of different STATUS types (e.g., "ORANGE", "APPLE", "LEMON" may belong to both a fruit type STATUS("ORANGE", "APPLE", "LEMON") and a flavor type STATUS("ORANGE", "CHOCOLATE", "VANILLA", "LEMON")). Consequently a subsequence such as "ORANGE".."LEMON" would be ambiguous.

6. Pointer Variables (D6).

a. Degree of compliance: F

b. Lacking a pointer facility, CS-4 fails to satisfy this requirement (it may be noted, however, that the data abstraction facility enables the user to define modes that capture some of the essential properties of pointers).

c. Modifying the language to add pointers would be a major effort,. due to interactions between pointers and other features such as storage allocation and type equivalence.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

   a. Degree of compliance: P

   b. CS-4 satisfies this requirement to a large extent, particularly in its data abstraction facility [4, Chapter 6]. The only area of disagreement is that CS-4 does not allow the user to extend the meanings of infix operators to operands of user-defined modes or to define new infix operators. It should also be pointed out that a shorthand for mode-invocations (a form of renaming that does not create a new mode) would be useful, for notational convenience.

   c. The modifications required to completely meet the requirement are substantial. An operator definition facility would have to be added to the language to allow the definition and extension of infix operators.

2. Consistent Use of Types (E2).

   a. Degree of compliance: P

   b. Although the CS-4 data abstraction facility helps remove some of the differences in use between built-in and programmer-defined modes, several differencs remain:

      (1) No component selection outside the mode definition is possible with user-defined modes. For the language to comply with E2, component selection would have to be invokable via either subscripting or dot qualification, and new values would have to be assignable to selected components.

      (2) Some of the built-in modes have prefix or infix operators associated with them. Since CS-4 lacks a facility to define new operators or extend existing ones, no user-defined mode can have any associated prefix or infix operators.

      (3) The "*" form is permitted in ARRAY mode-invocations but may not appear in invocations of user-defined modes.

(4) There are different object construction conventions for built-in vs. user-defined modes. For example, the mode-invocations for many of the built-in modes may be used as explicit generic conversion routines; such a facility is not present with user-defined modes.

c. Attempting to meet this requirement would entail major revisions to the language design. In fact, it is not clear that the state-of-the-art in language design is such that this requirement can be completely satisfied.

3. No Default Declarations (E3).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. As stated in [4, p. 2]: "the language does not provide any default declarations for the mode of an object."

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. CS-4 contains no provision for extending existing operators to new data types.

c. The required modifications would be non-trivial. As mentioned under E1, an operator definition facility would have to be added to the language.

5. Type Definitions (E5).

a. Degree of compliance: T

b. CS-4 satisfies this requirement via its data abstraction facility. A MODE definition specifies both a representation (parameters to the MODE and constants and variables defined within the mode-body [4, p. 158]) and a set of routines specific to the MODE. Since a user-defined mode is automatically different from the mode of the underlying representation, there is no way for routines applicable to the latter to be inherited by the former. In fact, not even all the routines that are defined within a mode-body are usable outside; the CAPABILITY attribute [4, p. 162] provided in the mode definition defines which of the

routines are so usable. As required by E5, CS-4 permits the user to define constructors, selectors, predicates, and type conversions; however, each of these must be invoked via explicit procedure invocations.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. Definition by enumeration of literal names is realized in the STATUS mode [4, Section 3.3]; Cartesian products are represented by ARRAY and STRUCTURE modes [4, Sections 3.4 and 3.5]; discriminated union exists in the form of the UNION mode [4, Section 3.7]; and the SET mode [4, Appendix B] captures the notion of the power set of an enumeration type.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement to a large extent. First, type definition by subsetting is not permitted. Second, although type definition by free union is possible (via MSTRUCTUREs [4, Section 8.1] or by disabling tag checking for UNION objects [4, Section 3.7.3.1]), programmers must possess special authority to achieve this effect [4, Section 7.2.2].

c. It is impossible to completely satisfy this requirement without conflicting with other requirements because of inconsistencies in the Tinman. Eliminating MSTRUCTURES from CS-4 would violate requirement J4, and always requiring run-time tag checking on UNION objects would violate J1, the requirement for efficient object code. In particular, it is desirable to eliminate run-time tag checking in the body of CASE statements where the tag was used to select the appropriate case.

8. Type Initialization (E8).

a. Degree of complaince: T

b. CS-4 completely satisfies this requirement, via the user's ability to define INIT (initialization) and TERM

(finalization) procedures as part of a data abstraction [4, pp. 165-167]. The INIT procedure is invoked implicitly at object allocation time; TERM is invoked implicitly on deallocation.

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. First, since the allocation rules follow a stack discipline, it is impossible for an object to be accessible unless it is also allocated (i.e., an object is deallocated by the system, not by the programmer, and this can occur only when the object is no longer accessible). Second, the provision of STATIC and AUTOMATIC storage classes [4, p. 20] allows the user to distinguish between access and allocation scopes. For an AUTOMATIC variable, these scopes are identical. For a STATIC variable, the allocation scope is, in effect, the whole program, but the access scope is the block in which the declaration appears.

2. Limiting Access Scope (F2).

a. Degree of compliance: T

b. CS-4 completely satisifies this requirement, primarily via its ACCESS directive [4, Section 7.3.1], which enables a program to make use of information defined in separately compiled programs. The ACCESSed program makes available via a CAPABILITY list a set of entities (variables, constants, modes, or routines) whose definitions are known within the program; the ACCESSing program specifies those entities that it requires, and it may rename them to avoid name conflicts. It should also be pointed out that CS-4's data abstraction facility provides "the ability to limit the access to separately defined structures;" the mode definition's CAPABILITY list specifies the available routines.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely. The scope of identifiers is always determined at compile-time, and declarations must precede references ("all declarative statements in a statement list must precede all executable statements in the same list" [4, p. 14]. Although the

exception handling facilities of CS-4 involve dynamic scope search rules for determining signal handlers [4, Section 5.3.3], the identifier names involved still have statically established scopes.

4. Libraries Available (F4).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement, via the ACCESS directive. However, there are some drawbacks to this facility that prevent it from achieving all the goals of F4. The following description appears in [4, p. 177]:

> Accessed entities behave as if their actual declaration had occurred at their point of access. Access of an entity whose declaration depends upon another entity which is not made available via the capability list attribute in the header of the accessed program will constitute a compiler-detected error.

Under these rules, program modularization can become quite tricky. As an example, suppose a program is to be written that will create, retrieve, and update data in some data base. A direct approach is to encapsulate in one program, say P, the definition of the data base (STATIC VARIABLE declarations) and the definition of routines that massage it (PROCEDUREs), but to promote only the routines. In this fashion, the main program Q would access P to interface with the data base. Since the representation of the data base had not been promoted, Q would be restricted to performing only "safe" operations on the data.

c. Unfortunately, the rule quoted above would force the promotion of P's VARIABLE declarations as well as its routines, since the variables comprising the data base representation would naturally be used as free-variables inside the routines. Thus, either "unsafe" access to the representation would be granted, or a different form of program modularization would be required.

d. In addition to the modularization problem, the rule as stated in [4] gives a misleading impression in its first sentence that a macro-like facility is being applied. If this were the case, then ACCESSing a variable would cause a

copy of the variable to appear in the ACCESSing program -- this is not the intended behavior, however.

e. The modifications required both to the language and the implementation to clean up the ACCESS mechanism are of a fairly minor nature. The rule regarding accessed entities would have to be changed so that entities that are not promoted via the capability-list attribute, while not directly available to the accessing program, would still be available to entities in the accessed program that were promoted (rather than causing a compiler-detected error).

5. Library Contents (F5).

a. Degree of compliance: T

b. CS-4 completely satisfies this requirement. First, anything definable in the language may be ACCESSed in a (separately-compiled) program. Second, CS-4 allows the use of external procedures [4, Section 8.3] (i.e., procedures whose bodies are written in other languages).

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: T

b. CS-4's ACCESS facility can be used to build both libraries (containing data objects and routines) and compools (containing mode definitions); thus, libraries and compools are indistinguishable.

7. Standard Library Definitions (F7).

a. Degree of compliance: PT

b. CS-4 goes quite far in satisfying this requirement. The primary facility for achieving "standard machine independent interfaces to machine dependent capabilities" is CS-4's Operating System Interface. As stated in [5, p. v]: "The purpose of the operating system interface is to standardize the interface between CS-4 programs and the variety of target machine operating systems which provide common functions in non-standardized forms." Other facilities in CS-4 for dealing with machine dependencies include the MSTRUCTURE mode (for defining physical storage layout), a direct code capability, an external procedure facility, and an environmental inquiry mechanism.

c. Although the CS-4 Operating System Interface comes close to satisfying this requirement completely, there are some areas in which the definition of the facilities provided is incomplete or vague. For example, nowhere is it specified whether or not EVENT is a mode, nor is it specified whether or not an update block (i.e., a critical region) can contain operations that cause the process to be blocked.

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: PT

b. CS-4 provides most of the control structures specified in this requirement. Sequential execution is the normal flow of control. Conditional execution is realized by the if-statement and case-statement [4, Sections 4.5 and 4.6]. Iterative execution is achieved in the repeat-statement [4, Section 4.7]; exception and asynchronous interrupt handling are achieved in the SIGNAL [4, Section 5.3] and event [5, Section 4.2] facilities. The only control structure required in G1 that is not contained in CS-4 is the ability to define or invoke recursive procedures. As stated in [4, p. 140]: "recursive calls (calls on a procedure while there still exists an active invocation of the same procedure) are not permitted."

c. The scope of modifications needed to add recursion to the language are discussed under requirement G5.

2. The Go To (G2).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely. The only qualification is that in CS-4 the GoTo statement may transfer control out of a BEGIN block in which it is enclosed [4, p. 124 and 135-136].

c. Only very minor modifications would be required to the language and the implementation in order to restrict the GoTo to the local scope level.

3. Conditional Control (G3).

a. Degree of compliance: P

b. The if-statement and case-statement of CS-4 partially satisfy this requirement. Differences that arise are (1) CS-4's conditional control operations are not "fully partitioned" (the if-statement need not contain an ELSE clause), and (2) there is no "general form of conditional which allows an arbitrary computation to determine the selected situation."

c. The modifications needed to require an ELSE clause on an if-statement are extremely minor. Although a general form of conditional such as Zahn's device adds no power to the language, it could be added without great difficulty.

4. Iterative Control (G4).

a. Degree of compliance: PT

b. The repeat-statement of CS-4 [4, Section 4.7] satisfies this requirement to a high degree. As specified in G4, the only loop control variable is local to the loop, entry is permitted only at the head, the common special case of a fixed number of iterations is handled efficiently, and the termination condition may appear anywhere in the loop through the use of an exit statement. Differences between CS-4 and G4 are as follows:

(1) The syntax for the fixed-number-of-iterations case is somewhat clumsy (to have a loop repeated, say, 10 times, the user must specify
FOR INTEGER (RANGE: 1 THRU 10) REPEAT...END or some other integer-mode-invocation denoting a value space with 10 elements).

(2) The case in which the number of iterations is fixed, but known only at run-time, cannot be handled with a for-phrase; instead, a while-phrase must be used, implying that a non-local loop-control variable will be available after loop-exit.

c. Modifying the for-phrase to permit a run-time bound and cleaning up the syntax are small changes to both the language and the implementation.

5. Routines (G5).

a. Degree of compliance: F

b. Lacking recursion, CS-4 fails to satisfy this requirement.

c. The modifications required to the language to add recursion are fairly minor. The programmer should be explicitly required to indicate which procedures are

recursive. This not only aids in readability but also aids the implementation in determining when an illegal (i.e., unintended) recursive call is made, and in enforcing the restriction that procedures cannot be defined within the bodies of recursive procedures. The changes required to the implementation are more substantial but are simplified by the fact that the dynamic storage allocation mechanism needed for recursive procedures is already required in CS-4 because of such features as dynamic array bounds at block entry.

6. Parallel Processing (G6).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement to a large extent. The only differences lie in the generality of CS-4 vs. the restrictions recommended (for efficiency reasons) by G6; e.g., CS-4 allows the definition of routines within the body of parallel routines that can have multiple simultaneous activations.

c. The main aspects of CS-4's parallel processing facility are as follows:

(1) Process management.

(a) Processes in CS-4 are created by declaring variables of mode PROCESS. A SCHEDULE_PROCESS command associates a program-level procedure with the process and indicates that the process is to be scheduled. The actual scheduling of processes is determined by user-specified priority levels or, for time-critical processes, by user-specified real-time constraints [5, p. 21, 25].

(b) The storage class for PROCESS objects is not restricted. Therefore, processes can be declared AUTOMATIC; these will be initiated dynamically. Nevertheless, because there is no recursion in the language, the maximum number of processes that can be created is known at compile-time.

(c) General capabilities are provided for

process-state inquiry, process interruption and resumption, process termination, and guaranteeing that an executing process will not be pre-empted [5, pp. 21-29].

(2) Shared data.

(a) A SHARED storage-class is provided for data objects that are to be accessed by several processes.

(b) Structured critical regions called UPDATE-blocks are provided to prevent disorder and deadlock that could result from concurrent accesses to shared data. All objects declared with the storage-class-attribute SHARED(PROTECTED) are only allowed to appear within such critical regions [4, p. 20, 127].

(c) A "corelink" capability is provided for the communication of data between processes having unrelated compilations [5, p. 33].

(3) Synchronization.

(a) Synchronization between processes is achieved by EVENT variables which can be SET by one process and WAITed on by other processes [5, p. 31].

(b) A user-specified connection can be established between EVENTs and implementation-defined hardware conditions. The EVENTs will be SET on the occurrence of these conditions and thus allow responses to hardware interrupts to be programmed in the high-level language [5, p. 32].

(c) Process delays based on a real-time clock are provided by the TIME_WAIT function [5, p. 67].

d. The modifications required to the language and the implementation to provide the restrictions required in G6 are of a minor nature.

7. Exception Handling (G7).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement fairly well through its signalling facility [4, Sections 4.11, 4.12, 4.13, 5.3.1], which provides a means of communicating the occurrence of an exceptional situation to user-specified handlers for that exception. Signal-handlers may be passed information via parameters, and they have rules consistent with those for normal procedures. Handlers may (1) handle the exception and return control to the point where the signal occurred; (2) decide the exception cannot be handled, take some other action, and transfer control to a point in the handler's immediate external environment; or (3) issue another signal.

c. Signals may be generated by explicit statements written in the program; in addition, they will automatically be generated for errors detected by the hardware (e.g., division by zero) or by compiler-supplied software checks (e.g., range and subscript bounds errors). The names of signals associated with errors are available to the user so that each program can define its own error-handling routines [4, Appendix F].

d. The only difference between CS-4 and G7 is that the CS-4 signal handling facility provides a slightly more general capability than that required by G7 and that the specification of the signal handler is not done through an exception handling formal parameter class.

e. Though not directly related to G7, there is a somewhat subtle interaction worth noting between CS-4's exception handling and data abstraction facilities. The problem is that the dynamic chain search rules prevent a single handler from handling a signal generated by disjoint procedures occurring within a MODE (or at program level). The reason for the problem is that at the time the signal is generated there would be no currently-active procedure that immediately contains the handler (the handler is defined local to a mode and not a procedure).

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. CS-4's Operating System Interface contains facilities that meet many of the requirements of G8. EVENT variables [5, Section 4.2] permit delay until some specified situation has occurred. If the delay is to be determined by a known time interval, then time-critical processes may be used [5, pp. 24-25]. Specification of relative priorities among parallel control paths is realized in the SCHEDULE_PROCESS command for normal (i.e., non-time-critical) processes [5, p. 26]. EVENT variables may be connected to real-time clock or asynchronous hardware interrupts [5, p. 66]; however, this is a separate facility from the (synchronous) exception handling mechanism described in G7. The synchronization of parallel processes may be achieved via events, corelinks [5, Section 4.3], or protected variables and update blocks (See G6).

c. Problems in CS-4 with respect to G8 come not so much from failure to satisfy specific requirements (although it might be pointed out that CS-4 lacks a convenient facility for a process to terminate itself) as from the occasionally incomplete nature of the description. For example, despite the attention paid to EVENT variables, nowhere is it mentioned whether EVENT is in fact a mode, or how latched EVENT variables are initialized. Also, it is not clear why only pulsed events should be connectable to external conditions. It should be mentioned, too, that having procedures such as SCHEDULE_PROCESS and SET_EVENT return a STATUS value causes a fair amount of notational clumsiness; one must either declare an otherwise unnecessary variable of the given type, or else embed the procedure in a conditional statement.

d. A moderate amount of revision is required to "clean up" the Operating System Interface. This includes both making the specification complete and redesigning some features to avoid notational clumsiness.

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

   a. Degree of compliance: PT

   b. CS-4 satisfies most of the requirements of H1. The language is free format, allows mnemonically significant identifiers (up to 32 characters in length [4, p. 6]), uses conventional notation, and does not permit abbreviation of identifiers or key words. The main deviation from H1 is that the grammar (at least the one presented in [4]) is somewhat complicated and even ambiguous. In connection with this, however, it should be noted that the grammar contained in [4] is intended to facilitate the description of the language; the complexity is partially caused by the desire to capture in the syntax a large part of the semantic restrictions defined in the language.

   c. Although the semicolon serves as a separator rather than as an explicit statement delimiter, this causes no problems since CS-4 is a statement-oriented language, not an expression-oriented language.

   d. It should also be noted that although the *-form of subscript and bounds specification is in fact a unique notation for a special case, it is also a conventional form.

   e. The modifications required to meet this requirement are fairly minor. An unambiguous grammar, suitable for use as an implementation grammar, has been devised for CS-4.

2. No Syntax Extensions (H2).

   a. Degree of compliance: T

   b. CS-4 completely satisfies this requirement.

3. Source Character Set (H3).

   a. Degree of compliance: P

   b. CS-4 partially satisfies this requirement. A major difference is that the CS-4 character set, based on full ASCII, contains elements outside the 64-character ASCII subset, with no defined transliteration for some of the

inaccessible characters. The characters outside the subset are vertical-bar, grave-accent, open- and closed-brace, and tilde. The lack of a substitute character for the vertical bar can be a serious problem, since the string concatenation operator is composed from this symbol.

c. A very minor amount of modifiction would be needed to define translations from the current character set to the 64-character ASCII subset. Only the lexical analyzer would be affected by the change.

4. Identifiers and Literals (H4).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement nearly completely [4, pp. 6-8]. The underscore is the break character for identifiers, but there is no break character for literals.

c. The modifications to add a break character for literals would be trivial. Only the lexical analyzer would be affected. The main difficulty, however, is in choosing a suitable character. There is no generally accepted break character for literals and none of the likely candidates (e.g., blank, underscore) is clearly superior to the others and each presents problems with readability and error detection.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. In CS-4, there is no way to continue lexical units across lines, but the language fails to satisfy the part of H5 that requires a means for including end-of-line characters in string literals.

c. The modification required to completely meet the requirement involves only a small change to the lexical analyzer.

6. Key Words (H6).

a. Degree of compliance: P

b. CS-4's treatment of reserved words [4, Appendix E] conforms partially to this requirement. Symbols in CS-4 are grouped into three categories: (1) those whose meanings are fixed (reserved); (2) those whose meanings may be assigned freely by a program but which have fixed meanings in special contexts independent of programmer-assigned interpretations; and (3) those whose CS-4 meanings are lost (overridden) when new meanings are assigned by programs (e.g., built-in functions). The keywords referred to in H6 correspond to all the category (1) and (2) key words in CS-4 and a few of the category (3) key words. All the category (1) key words satisfy H6 with one exception. TRUE and FALSE, which are literals of MODE BOOLEAN, are useable in identifier contexts.

c. The modifications required to meet H6 are fairly minor. Category (2) would have to be eliminated and all the category (2) key words put into category (1) (i.e., reserved). A few of the category (3) key words would also have to be added to category (1) (i.e., AND, OR, NOT, NAND, NOR, EQV, XOR, IMP). The Boolean literals TRUE and FALSE should be changed to .TRUE and .FALSE to avoid confusion with identifiers.

d. It should be noted that although CS-4 has a large number of key words (categories (1) and (2) have about 100), eliminating many of them would change the nature of the language and defeat some of the language's goals (e.g., readability and reliability through useful redundancy).

7. Comment Conventions (H7).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement [4, pp. 9-10]. H7 requires that the language provide only one comment convention. CS-4 provides two comment forms: one version (where the comment is delimited by percent-sign and end-of-line) fully meets the provisions of H7; the other form (delimited by braces) enables arbitrary program segments to be easily turned into a comment.

c. This requirement can be met simply by deleting the second comment form from the language.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. CS-4 satisfies this requirement completely.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. CS-4 satisfies this requirement only to a limited extent. The main example of uniform referent notation in CS-4 is that ARRAY subscripting and procedure invocations share some common forms. However, the *-form is available for ARRAY subscripting and not as a procedure argument, and key-word parameters are available for procedures but not for subscripting. The STRUCTURE mode violates uniform reference, since the dot qualification form used for data reference is not a valid function call. In addition, the absence of facilities for user-defined functions to be invokable in write-contexts (e.g., as the target of an assignment) implies that, with the exception of some built-in functions such as SUBSTR, forms appearing in such contexts will be data-references. The lack of a uniform referent notation is even more apparent for data abstractions, since no data reference form is available for selecting object components.

c. Modifications to the language to meet this requirement would be substantial and would, in effect, change the nature of the language. The dot qualification form of STRUCTURE can be fairly easily changed so that, for example, instead of writing S.A, one would write A(S). Similarly, keyword parameters can be eliminated from procedures. The other changes, however, are much more major. Eliminating the *-form of array subscripting would seriously hamper the power and convenience of the language. Allowing user-defined functions in write-contexts would require sizable design effort, especially to establish suitable restrictions. With regard to data abstractions, determining a uniform referent notation is currently a research topic and not within the state-of-the-art.

10. Consistency of Meaning (H10).

a. Degree of compliance: T

b. CS-4 complies with this requirement completely. In particular, the error-prone features cited in H10 (use of = to denote both assignment and equality, special interpretation for parenthesized arguments) are absent from CS-4.

Section I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

   a. Degree of compliance: P

   b. The CS-4 language specification suffers from a problem common to many language reference manuals, which brings it into conflict with I1. The difficulty lies in the inconsistency with which the behavior of erroneous programs is treated. In many cases, the language defines run-time conditions that are signalled when errors occur [4, Appendix F]. However, there are many more instances in which rules are specified that do not define the program behavior when the rules are violated: e.g., the restrictions governing update blocks [4, p. 128], go-to statments [4, p. 136], or even what happens with programs containing syntax errors. The result of such an absence of specifications is, as stated in I1, "implementation-dependent defaults with the translator determining the meaning of programs."

   c. A moderate amount of revision is necessary to the language specification to explicitly state the program behavior when rules are violated. Such specification could be achieved by a formal definition of the language, although the latter would be a major undertaking.

2. Object Representation Specifications Optional (I2).

   a. Degree of compliance: P

   b. CS-4 partially satisifies this requirement. With respect to data representations, the programmer may specify storage formats only via the MSTRUCTURE mode; in this sense, he may override the default representation chosen. With respect to subroutine calls, the default is closed (i.e., out-of-line) compilation, but this may be overridden by the programmer. There is no programmer control over reentrant vs. nonreentrant code generation; this decision is implementation dependent. A complete list of compiler-supplied defaults in CS-4 appears in [4, Appendix D].

c. The modifications required for the language to give the programmer explicit reentrant code generation are minor. Moderate modifications might be required to an implementation in order to generate reentrant code, and there are potential conflicts with efficiency if the hardware does not support reentrant procedure calls.

3. Compile-time Variables (I3).

a. Degree of compliance: PT

b. CS-4 almost completely satisfies this requirement. CS-4 has an environmental inquiry facility. In addition to language defined constants, this facility also has machine-dependent constants which can be interrogated inside a program [4, Section 8.4]. In addition, the language provides for machine-dependent machine-names and storage-unit-names for specifying the machine configuration in MSTRUCTURE definitions. The only way in which CS-4 fails to meet I3 is that environmental inquiry is not specified for such facilities mentioned in I3 as operating system, peripheral equipment, and special hardware options.

c. The modifications required to include additional environmental inquiry constants to the language are minor.

4. Conditional Compilation (I4).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement. There is no compile-time conditional facility, but the language does contain an environmental inquiry capability [4, Section 8.4] and allows some expression evaluation at compile-time.

c. Substantial modifications would be required both to the language and the implementation to add a conditional compilation facility. A significant amount of design would be required to select the appropriate forms of compile-time statements (e.g., CTIF, CTCASE), compile-time variable definitions and compile-time expressions. A non-trivial amount of effort would also be needed to implement such facilities.

5. Simple Base Language (I5).

a. Degree of compliance: P

b. CS-4's base language is not very simple, (comprising almost all of [4] and [5]); moreover, the reference manual does not distinguish between base language and extension-obtainable constructs. However, the latter are identifiable implicitly, consisting of many of the built-in procedures for the primitive modes (e.g., [4, Appendix C]).

c. One aspect of CS-4 that contributes to its lack of simplicity is the inclusion of COMPLEX, VECTOR and MATRIX as built-in modes with built-in operators. This was done for notational convenience since the language lacks an operator definition facility. The major complexity of the language, however, comes from the data abstraction facility [4, Section 6] which is specifically needed to meet requirements E1 and E5. A large part of this complexity comes from interactions with other features of the language. For example, numerous restrictions are needed on the kinds of parameters that can be used when explicitly re-defining one of the standard operations for a user-defined mode (e.g., the first formal parameter to an ASSIGN routine may not be bound by COPY [4, p. 164]).

d. It is extremely difficult to modify a language to meet a requirement as general as this. Making a language simple involves the deletion of numerous features. However, as will be discussed in Section XV of this chapter, there are very few features in CS-4 that are not needed to satisfy requirements in the Tinman.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. CS-4 partially satisfies this requirement, in that the language places a limit on the length of identifiers (viz., 32). However, CS-4 specifies neither the maximum number of array dimensions, the maximum level of parenthesis nesting, nor the maximum number of identifiers in programs.

c. It would be a minor modification to the language and to a well-structured implementation to specify langauge-defined limits for the number of array dimensions, level of parenthesis nesting, and the number of identifiers.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. Lacking restrictions of the kind described in I7, CS-4 completely satisfies this requirement.

Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. CS-4 has a wide variety of features that promote the production of efficient object code, but there are also some facilities that may incur run-time overhead even when not used. The following sub-paragraphs illustrate both issues.

(1) Features facilitating efficiency.

(a) The mode parameters RANGE (for integers and reals), PRECISION (for reals and fractions), and string-size can be used by the compiler to determine the minimum number of bits that must be allocated for the representation of an object's value. These parameters are not specified in terms of bits, but in terms of the (integer or real) range of the value space, the decimal digits of precision, and the number of characters, respectively [4, p. 37, 44, 59, 100].

(b) The user has explicit control over whether procedures pass copies of actual parameter values or simply reference those values [4, p. 144]. This control is independent of whether the parameters are intended for input only, output only, or both input and output.

(c) Procedures declared to be OPEN will have their bodies expanded inline at each point of call, thereby trading off space for reduced invocation times. In all other ways, the semantics of OPEN and normal CLOSED procedures are identical [4, p. 148]. Procedures containing assembly or machine language code can also be declared OPEN and expanded inline.

(d) A NORECALL attribute can be associated with procedures. It is used by a compiler in optimizing multiple procedure calls having the same argument values [4, p. 150].

(e) Checking-directives are available to disable the generation of run-time checking code [4, p. 178].

(f) Many of the language-supplied procedures and operators will be invoked at compile-time if all of their arguments or operand values are known at compile-time. The resulting values can be used not only in contexts requiring compile-time values, but also in contexts that normally would require run-time invocations [4, p. 173].

(g) Although conversions are possible using explicit calls on construction routines, implicit conversions resulting in unknown overhead are not permitted [4, p. 26, 28].

(h) Storage allocation may be specified as either AUTOMATIC (within name scope levels), STATIC (at program level), SHARED (between processes at program level), or ABSOLUTE (for machine-dependent objects of mode MSTRUCTURE) [4, p. 20]. Pointers and HEAP storage requiring garbage collection are not provided.

(i) Recursive procedures are not permitted [4, p. 140]. Therefore, AUTOMATIC data (but only the dope vectors for dynamic arrays and strings) may be laid out statically, with restricted scopes of access, and not require references through a display. The language specification uses the terms "allocate" and "deallocate" to indicate when the data's storage is dynamically made available or unavailable, and where routines are invoked for initiation (of values for data and states for processes) and termination.

(j) Efficiency of storage using overlaid data can be achieved in three ways:

by compiler-determined storage layouts for AUTOMATIC data [4, p. 20];

by compiler-determined storage layouts for objects of discriminated UNION mode [4, p. 113]; and

by choosing storage-unit-values such that component fields in machine dependent MSTRUCTURES overlap [4, p. 183].

(k) The user can control the efficiency of initiate and terminate routines using the OPEN attribute as discussed above. In addition, object declarations using language supplied modes can specify that the initiate routine is not to be invoked [4, p. 18]. The terminate routines for the language-supplied modes perform no action [4, p. 26]; therefore, they can be implemented as body-less OPEN procedures that require no overhead.

(2) __Features with hidden run-time expense__.

(a) Array bounds need not be known at compile-time and thus require a "dope vector" in the general case. However, a dope vector will have to be used even when the bounds are known at compile-time (e.g., when the array is passed by reference as an argument to a routine whose formal parameter is an array with run-time determinable bounds).

(b) CS-4 has fairly powerful facilities in the area of exception handling and parallel processing. It is not obvious that run-time expense is avoided when these features are not used.

c. CS-4 comes very close to satisfying this requirement. In particular, the checking-directives mentioned above (b(1)(e)) give the programmer explicit control over efficiency versus reliability tradeoffs. Although there are a few features that may have hidden run-time expense (sub-paragraph (2) above), such features were designed to balance efficiency with reliability (possibly at the expense of simplicity), and could not be easily modified to completely satisfy this requirement.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: TU

b. This requirement is more a function of the translator than of the language itself, but it should be mentioned that there is nothing in CS-4 that prevents this requirement from being satisfied. For example, since the rules for expression evaluation result in side effects being carried out in left-to-right order, there is no danger of different translators producing object programs yielding different results for the same source program.

3. Machine Language Insertions (J3).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement almost completely via its MPROCEDURE direct code facility [4, Section 8.2]. The syntax and semantics of MPROCEDUREs (and their parameters) are consistent with those of normal procedures (e.g., they may be expanded inline). The rules are consistent, but not identical, because certain machine and implementation dependencies cannot be hidden: e.g., the machine-dependent register location of an actual parameter is specifiable, and may be used in the assembly code [4, p. 195]. A minor difference between CS-4 and J3 lies in the nature of the encapsulation of the machine language insertions. Instead of permitting direct code "only within the body of compile-time conditional statements" (as stated in J3), CS-4 employs MPROCEDURE...END brackets and requires the specification of a TARGET machine. Also, the direct code facility may only be used by programs possessing the MPROCEDURES authority [4, p. 194].

c. The modifications required to completely satisfy this requirement are dependent on adding compile-time conditional statements to the language (see I4).

4. Object Representation Specifications (J4).

a. Degree of compliance: PT

b. CS-4 satisfies this requirement fairly well, via the MSTRUCTURE mode [4, Section 8.1]. With this feature, the user can specify the order and width of fields, the presence of "don't care" fields, and data alignment; it is also possible to associate source language data (but not programs) with special machine addresses. The use of MSTRUCTURES is restricted in a similar fashion to

MPROCEDUREs; the program must possess the MSTRUCTURES authority [4, p. 180]. One facility that CS-4 lacks in the area of object representation specifications is the ability to indicate a degree of packing for non-MSTRUCTURE data.

c. Modifying the language to allow a packing attribute on any data declaration is a relatively minor change. However, the effect on the implementation is non-trivial and can cause a great deal of run-time overhead for unpacking and implicit representational conversions. For example, it is non-trivial to implement by-reference parameter passing for a single Boolean variable in a packed array of Booleans.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: T

b. CS-4 provides OPEN and CLOSED as compilation class attributes for procedures [4, p. 148].

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: T

b. None of the features in the CS-4 language itself [4] requires the existence of an operating system. CS-4 has an Operating System Interface [5] to make available to the language existing operating system capabilities in a standard fashion. Should a particular capability (or an entire operating system) not exist, they can be implemented directly in the OSI. It should be noted, however, that certain language features specifically required by the Tinman (e.g., parallel processing, G6) require run-time support (generally known as an operating system).

2. Program Assembly (K2).

a. Degree of compliance: P

b. The ACCESS mechanism of CS-4 partially satisfies this requirement.

3. Software Development Tools (K3).

a. Degree of compliance: U

b. This issue is not addressed in the language manual.

4. Translator Options (K4)

a. Degree of compliance: U

b. This issue is not addressed in the language manual.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: FU

b. Although CS-4 provides no facilities (except comments) for assertions, etc., the language has nothing in it that would prohibit these forms. It should be noted, however, that interpretation of assertions, etc. (specified as implementation-dependent in K5) has a potential interaction with exception handling.

Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

2. No Subset Implementations (L2).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

3. Low-Cost Translation (L3).

   a. Degree of compliance: P

   b. CS-4 partially satisfies this requirement. However, the size and complexity of the language may interfere with low-cost translation. Modifications to the language to meet this requirement are probably not desirable because conscious design trade-offs were made in the language in favor of such goals as reliability and efficient object code over fast translation.

4. Many Object Machines (L4).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

5. Self-hosting Not Required (L5).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

6. Translator Checking Required (L6).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

7. Diagnostic Messages (L7).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

8. Translator Internal Structure (L8).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

9. Self-Implementable Language (L9).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

   a. Degree of compliance: P

   b. All the language features in CS-4 are within the state-of-the-art with the possible exception of the data abstraction facilities. Although some existing languages (e.g., SIMULA-67) support data abstraction, this is an on-going research area. It should be noted, however, that such facilities are specifically required in E1 and E5.

2. Unambiguous Definition (M2).

   a. Degree of compliance: P

   b. Although CS-4's semantics have not been defined formally, an attempt has been made to be rigorous and complete. However, as has been noted under specific requirements, there are places where the language manual is incomplete, especially with regard to the Operating System Interface. On the whole, the manual is well-organized and quite readable, and although not quite suitable as a user introduction to the language, it does much better in this regard than most language reference manuals. Its main failing comes from the manner in which features are defined. Rather than explaining the capabilities of features as would be done in an introductory document, the reference manual frequently states descriptions in the form of restrictions.

   c. A moderate amount of revision would be required to ensure that the language is defined completely and unambiguously.

3. Language Documentation Required (M3).

   a. Degree of compliance: U

   b. The CS-4 Language Reference Manual [4] is not intended as introductory-level user documentation, although in that regard it is better than many other language defining documents.

4. Control Agent Required (M4).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

5. Support Agent Required (M5).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

6. Library Standards and Support Required (M6).

   a. Degree of compliance: U

   b. This issue is not addressed in the language manual.

Section XV. CONCLUSIONS REGARDING CS-4

1. Objectives of CS-4 and Tinman.

CS-4 comes extremely close to meeting all the objectives of the Tinman. CS-4 is a recently designed language which incorporates the latest advances in HOL technology as well as features that support current program design methodologies.

2. Summary of Major Areas of Conflict between CS-4 and Tinman.

a. Data and Types. There are several major conflicts between CS-4 and Tinman. CS-4 does not have a fixed-point data type. In addition, CS-4 does not permit user-defined character sets.

b. Operations. There are no really major conflicts but there are a number of minor ones.

c. Expressions and Parameters. There are several conflicts in this area. CS-4's parameter rules for modes are not consistent with parameter rules for procedures. CS-4 does not have precisely the four kinds of parameters listed in the Tinman. CS-4 does not have the generic procedure capability called for, nor does it allow a variable number of parameters to procedures.

d. Variables, Literals, and Constants. The major conflict in this area is that CS-4 does not have pointers. In addition, CS-4 does not permit subsequences of enumeration types.

e. Definition Facilities. The major conflict in this area is that CS-4 does not permit operator definitions. In addition, user-defined types are not indistinguishable from built-in types.

f. Scopes and Libraries. There are no major conflicts in this area.

g. Control Structures. The major conflict in this area is that CS-4 does not have recursive procedures. There are, in addition, numerous minor conflicts.

h. Syntax and Comment Conventions. The major conflict in this area is the clumsiness of the mode declaration syntax. Smaller conflicts include CS-4's ambiguous grammar, unreserved keywords, use of a larger character set than 64 character ASCII, and variances from uniform referent notation.

i. Defaults, Conditional Compilation, and Language Restrictions. The major conflicts in this area are CS-4's lack of a conditional compilation facility and the fact that the behavior of erroneous programs is often unspecified.

j. Efficient Object Representations and Machine Dependencies. There are no major conflicts in this area. Minor ones include the fact that some CS-4 features may incur run-time overhead even when not used, and the fact that packing can only be specified in MSTRUCTURES.

k. Program Environment. The major conflict in this area is that CS-4 does not have a special facility for including assertions in programs.

l. Translators. The major conflict in this area is that CS-4 has features which make low-cost translation difficult.

m. Language Definition, Standards and Control. The major conflicts in this area are the fact that CS-4 data abstraction facilities are probably not within the currently accepted state-of-the-art and the fact that there are places where the language is incompletely specified.

3. Unnecessary Features in CS-4.

The unnecessary features in CS-4 (in the sense that they are not needed to satisfy the Tinman requirements) are the arithmetic aggregate modes (COMPLEX, VECTOR, and MATRIX), keyword parameters, and the SHARED(UNPROTECTED) attribute. The arithmetic aggregate modes were built in for notational convenience but could be removed if operator extension were added to CS-4. Keyword parameters also provide notational convenience and we recommend their retention. The SHARED (UNPROTECTED) attribute is redundant -- i.e., it is equivalent to STATIC -- and should be removed.

4. Recommendations concerning CS-4.

On the basis of the evaluation conducted in this chapter, we conclude that CS-4 comes very close to meeting the Tinman, both in its goals and in the specific requirements. CS-4 can be modified, sometimes quite trivially and sometimes with more difficulty, to meet most of the Tinman's requirements. There are some places, however, (noted throughout this chapter and in Appendix III) where we feel that it is the Tinman which should be modified.

CHAPTER 5

JOVIAL (J73/I) EVALUATION

Section I. LANGUAGE SUMMARY

1. Lexical Properties.

a. JOVIAL J73/I, called JOVIAL throughout this report, is a free-format language containing an explicit statement delimiter, ";". No special meaning is ascribed to end-of-line; in fact, all issues concerning it are ignored in [14].

b. The language is defined using 59 characters, all of which appear in the 64-character ASCII subset. Other:characters (*) (i.e., any an implementation wishes to support) may appear internal to character constants and comments.

c. JOVIAL names consist of arbitrary length sequences of letters,, digits and special symbols (prime and dollar-sign). Only the first 31 characters are significant. The first character of any name must be either a letter or $. The dollar sign may have an implementation-dependent meaning. The prime (') is included as a break character.

d. Comments may be of any length, are unaffected by end-of-line, and are delimited by quotation marks. A slight complication arises from the simultaneous use of quotation marks to delimit DEFINE strings (see compile-time facilities). To avoid ambiguity, comments may not appear within such strings. Except for the above restriction, comments may be placed between any two language symbols (lexical units).

(*) The colon is used in the JOVIAL definition, [14], to form multi-word non-terminals. Whenever colon-separated words appear within the body of this chapter, unless otherwise defined, they should be considered to denote a JOVIAL non-terminal.

2. Data Types.

The JOVIAL data types are categorized in Figure 4.

a. Scalars. The five JOVIAL scalar data types are floating point, signed and unsigned integers, and bit and character strings. The size for all scalar variables may optionally be specified, along with their declaration. If not so specified, strings default to a length of one and numeric variables to an implementation-dependent size. All size definitions, except for character strings, are made in terms of bits. A single character occupies a byte which is of implementation-dependent length.

(1) Behavioral properties common to all scalar data types. Assignment and relations are defined for all scalar data types. Implicit conversions will sometimes be made when necessary to force type compatibility. The rules of type matching permit conversions to occur in one direction in the type hierarachy -- character string (lowest), bit string, integer (signed and unsigned), floating point (highest). Explicit conversion routines between all scalar types are available.

The relational operators in JOVIAL are: =, <>, <=, >, >=. In comparisons between strings of differant size the shorter operand will be padded on the right for character data and on the left for bit strings. Bit strings are padded with zeros. The character padding element is not specified.

(2) Behavioral properties specific to numeric types.

(a) Arithmetic operators. JOVIAL supports a full complement of arithmetic operators; +, -, /, \ (modulo), and ** (exponentiation). The resulting type of a numeric:formula is usually that of the hierarchically higher operand (e.g., floating point + integer yields a floating point result). In this context, signed integers are placed above unsigned ones. The only exception to this rule is that exponentiation involving only integers will frequently yield a floating point result.

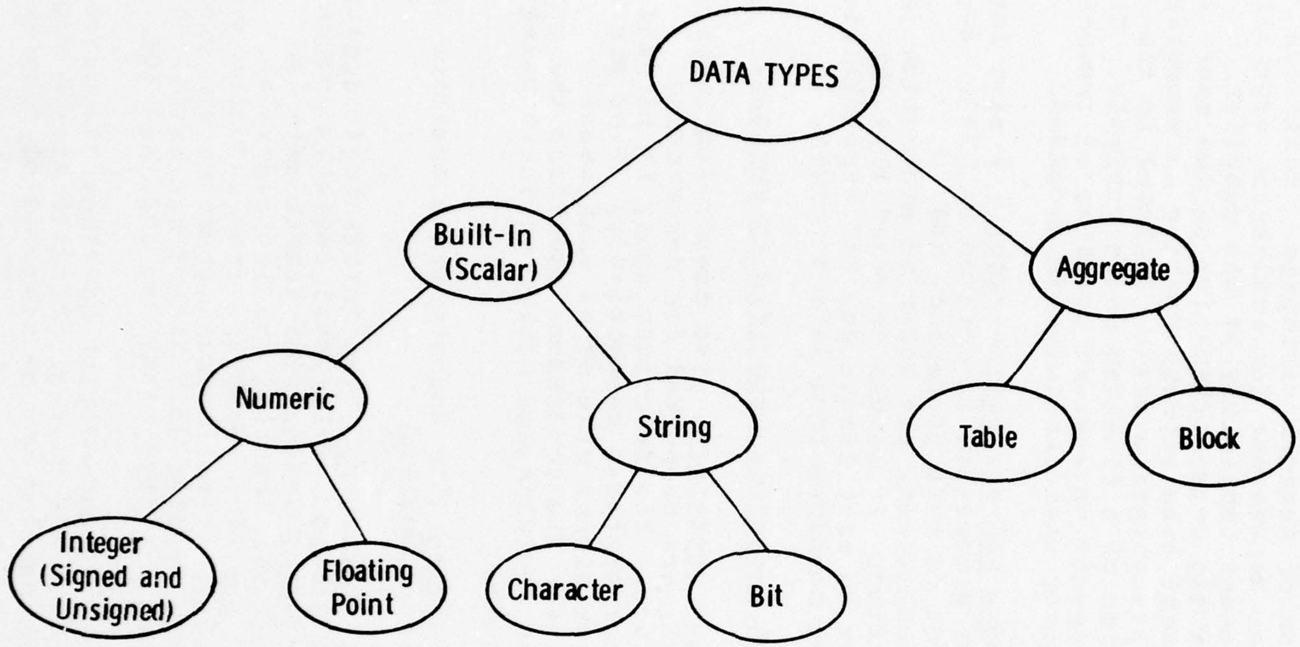

Figure 4. Data Types in JOVIAL

(b) Precision and scaling. Through size specifications the programmer can define the precision of numeric variables. Scale and precision management for arithmetic expressions is performed implicitly by the compiler. Overflow may cause truncation on the most significant bits without causing an exception. A truncation option may be included in the definition of a floating point variable. This option specifies whether rounding or truncation is to occur when precision is exceeded.

(c) Precedence and evaluation order. A nine level operator precedence is defined in [14]. Beyond adherance to this precedence and parenthesization, the order of evaluation is unspecified. It should be noted that, in contrast to most languages, the exponentiation operator combines from left to right.

(3) Behavioral properties specific to strings.

(a) Substring operators. Two pseudo-functions, BIT and BYTE, are supported for referencing substrings. The functions apply to bit and character strings, respectively. They may appear on either side of an assignment statement. The programmer specifies the starting position and number of units being referenced.

(b) Concatenation. No concatenation operator is supplied in JOVIAL.

(c) Logical operations. The JOVIAL logical type is the bit string. Relational formulas return a bit string of unspecified length with the rightmost bit denoting the logical value. A one in this position denotes a true relation and a zero false. Five logical operators are provided which are applied on a bit-by-bit basis. These operators are: NOT, AND, OR, XOR, and EQV.

b. Aggregate Data Types. JOVIAL provides two aggregate data structures, tables and blocks. A table may be viewed as an array in which entries can be composed of several

scalar items. Blocks constitute a general structure in which any set of data objects (scalar or aggregate) may be grouped. The only operations supported for aggregate data types are component selection and parameter passing. Table declaration is quite complex, containing many different options. The programmer may define ordinary and specified tables. An explicit object representation is included in the declaration of a specified:table. Management of ordinary:table representation is done by the compiler with the programmer supplying packing density and allocation order. Packing densities are Non-packed, Medium packed, and Densely packed. The allocation order may be parallel or serial. Allocation order has meaning only for multi-item table entries. Serial allocation connotes that all words for a given entry appear consecutively in memory. In a parallel table all words for item 1 are grouped together, followed by those for item 2, etc. Specified tables permit the programmer to define the exact object representation of his data. Each item description of a specified table includes the word and bit position of that item in an entry. Item fields may overlap. The number of words per entry must be declared, but this is only used for indexing purposes and a given entry may exceed this number. In the case of variable length entries, the programmer is responsible for determining the proper index for locating a desired record.

3. Procedures.

JOVIAL supports both procedures and functions. A procedure may have any number of input and output parameters and is invoked by a call:statement. Functions may only contain input parameters, are invoked by the appearance of their name within a statement, and return a value of any scalar type.

a. Parameter Types. JOVIAL parameters are differentiated by the object type of the argument.

(1) Item parameters. A scalar data item may be declared as an input parameter, output parameter or both. Actual item input parameter values are copied into their associated formal parameters upon procedure entry. If an item is a formal output parameter, its value is copied into the proper actual parameter upon procedure exit.

(2) Table and block parameters. Aggregate data objects may only be declared as input parameters. They are passed by reference. No type-checking is done on these parameters. The associated memory area is treated as a bit-string and may be reorganized in any manner.

(3) Statement name parameters. A statement label may be declared as an input parameter. A GoTo within the subprogram body can cause a transfer of control to that label.

(4) Subprograms as parameters. Both functions and procedures may be passed as parameters. The attributes of such actual parameters must match those of the formal parameter declaration. The attributes include type and number of parameters and for functions the type of the result.

b. Recursion. JOVIAL does not directly support recursion. However, through the use of based procedures (see Storage Allocation) the programmer may explicitly simulate recursion.

4. Statements.

a. Null Statement. This permits extra semicolons to appear in programs.

b. Assignment Statement. The assignment statement in JOVIAL may have multiple targets. If several value destinations are specified, the assignments are made in left to right order. The type semantics of assignment were discussed in 2.a(1).

c. GoTo Statement. This statement allows control to be transferred to any label within an enclosing block or to any REFed label (see REF statement).

d. Return Statement. The RETURN statement when encountered causes procedure exit. An optional procedure name may be specified. The procedure so named must lexically enclose the RETURN and will be the one exited. Only the output parameters of the specified procedure will be assigned their proper values.

e. Stop Statement. Execution is terminated when this statement is reached.

f. Iterative Statements. Two forms of loop statements exist in JOVIAL.

(1) While statement. The WHILE loop executes a statement repeatedly until some logical condition becomes false.

(2) FOR Statement. A loop control variable must be included in the FOR statement. Clauses exist for initializing, incrementing and repeatedly replacing the value of the loop control variable. Termination is controlled by a clause identical to the WHILE statement.

g. IF Satement. The JOVIAL IF statement may have either one or two alternatives. Its syntax is somewhat unusual in that the word "then" does not appear (e.g., IF AA<Ø; AA=-AA;).

h. Switch Statement. This statement is equivalent to a numeric case statement. Depending on the integer value of a numeric:formula one of several labelled alternatives is selected. An optional default clause can be specified to be executed when no match is found. A command following a given case specifies that the succeeding case is also to be executed. The behavior when no match is found in the absence of a DEFAULT clause is undefined.

i. DEF Statement. A DEF statement promotes the availability of the names it includes to external modules. Any RESERVE variable (see Storage Allocation), statement, procedure, or function name may be DEFed.

j. REF Statement. The REF statement can extend the accessibility of any DEFed name to the scope of the REF.

5. Storage Allocation.

a. Variable Declaration. Three allocation methods are defined for variables, RESERVE, based and IN. RESERVE data is permanently allocated. Based variables require no storage memory. They are simply templates which may be laid upon any region of memory. Items declared to be IN are only

available within a given scope. Between entries to that scope their values become undefined. A subprogram may also be declared to be of one of the above types. The effect of such a declaration is that all variables declared within the procedure without an allocation:specifier are of the allocation type defined for the procedure. In addition, all other memory used by the routine (e.g., storage for a function result) is also allocated in the specified manner.

b. Static versus Dynamic Allocation. Since all item sizes and tables bounds are compile-time known and recursion is not permitted memory may be allocated statically in JOVIAL.

6. Process Scheduling.

JOVIAL contains no facilities for process scheduling.

7. Files and I/O.

[14] does not define any input or output facilities.

8. Exception Handling.

The only facilities existing in JOVIAL for exception handling are those the programmer implements through the use of statement name parameters. This mechanism is used to signal value changes caused by explicit conversion; however, no general facility exists for the purposes of exception handling.

9. Compile-Time Facilities.

JOVIAL contains fairly extensive compile-time facilities. Macros are supported by the DEFINE feature. Conditional compilation can be achieved by use of the !SKIP, !BEGIN, and !END directives. Compile-time variables are supplied. These include: BITSINBYTE, BITSINWORD, BYTESINWORD, and LOCSINWORD. The use of separate modules is encouraged by the !COPY directive (see [14, Section 6.1.1]), and REF and DEF statements (see 4.i and 4.j of this section). A compool facility is defined for the language.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. JOVIAL partially satisifies this requirement. JOVIAL requires that the type of each variable and each component of a composite data structure be explicitly specified in the source program. The language is not, however, strongly typed: on assignment or within expressions many required conversions are performed implicitly [14, Section 1.7.1.2]; the OVERLAY facility provides a free union allowing variables of one type to be used as if they had the characteristics of another type [14, Section 1.5]; subscripting a table to obtain an entry causes the resulting set of items to be interpreted as a bitstream [14, Section 4.11.2]; implicit conversions are performed when items are passed as parameters to procedures, but when tables or blocks are passed, neither implicit conversions nor type checking is performed [14, Section 2.2.3].

c. The modifications needed to satisfy this requirement are substantial. Adding strong typing to the language would drastically alter the nature and intent of the language. Type checking permeates so many features of the language (e.g., assignment, expressions, parameter passing) that attempting to add strong typing after the fact would require a major effort to ensure that all features affected have been dealt with. The modification required to the implementation would be major since type checking would have to be added throughout and implicit conversions removed.

2. Data Types (A2).

a. Degree of compliance: P

b. JOVIAL provides built-in data types for <u>integer</u>, <u>floating-point</u>, <u>character string</u>, and <u>bit string</u>. (Bit strings in JOVIAL are considered to be right adjusted -- when two bit strings of different lengths are involved in an expression, the shorter is padded on the left with zeroes [14, Section 1.7.2]. The JOVIAL character string on the other hand is left adjusted.) Objects having one of the built-in types are called <u>items</u> in JOVIAL. For aggregate data objects the language allows <u>tables</u> (arrays of entries,

where an entry is a set of items), and <u>blocks</u> (heterogeneous structures of items, tables, and other blocks). The main points of disagreement with the Tinman are the lack of a fixed point type and the fact that there is not an explicit character type, but rather character strings.

c. The scope of modications required is discussed under A4 (Fixed Point Numbers) and A5 (Character Data). As will be seen, the needed changes are extensive.

3. Precision (A3).

a. Degree of compliance: P

b. JOVIAL permits precision specification for individual variables [14, Section 2.1.3.2]. When two floating point variables in an expression have different precisions an implementation dependent padding of the lower precision value occurs [14, Section 1.7.2]. JOVIAL provides no mechanism for specifying global (to a scope) precision for floating point arithmetic.

c. The modifications required to the language and the implementation are relatively minor. A global precision declaration would have to be added to the language. The impact on the implementation would be a change to the declaration processor which enters information in the symbol table.

4. Fixed Point Numbers (A4).

a. Degree of compliance: F

b. JOVIAL does not offer fixed point numbers aside from integers. The integers are either signed or unsigned, and are treated as exact quantities with a step size of one.

c. The modifications required to add fixed point numbers to the language are substantial. A large design effort would be required both to design the fixed point features and to ascertain their interactions with other features such as the arithmetic operations. In addition, these modifications would interact with the modifications needed to add strong typing to the language (A1).

d. The impact on the implementation would be substantial, affecting all phases of compilation.

5. Character Data (A5).

a. Degree of compliance: F

b. Although JOVIAL provides the user with a mechanism for defining enumeration types, the character set is not defined as an enumeration type. [14] does not specify the order of characters; therefore the result of character relational expressions [14, Section 4.6.2] is entirely implementation dependent.

c. To modify JOVIAL to meet this requirement would be a major effort. To allow the user to define his own character set and also allow literals composed of strings of such characters is not desirable and not within the state of the art of language implementation. Even choosing a fixed character set (e.g., ASCII), it would be difficult to make characters behave like any other enumeration (STATUS is really just a way of assigning constant names to integers). Allowing characters to behave like STATUS values implies that fixed integer values must be assigned to character values and all integer operations will then be defined on characters (e.g., multiplication!). To prevent such a situation really requires the redesign of the STATUS mechanism.

6. Arrays (A6).

a. Degree of compliance: P

b. JOVIAL requires user specifications of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. JOVIAL requires both upper and lower bounds to be fixed at compile time. The JOVIAL enumeration type (STATUS constants) has an explicit user-defined mapping onto the integers [14, Section 2.4]. Array dimensions may be declared with default lower bounds (equal to zero if not specified) and upper bound explicitly specified [14, Section 2.1.5.3.1]. The value of a type declared by enumeration may be used for array reference -- as any integer value might be [14, Section 4.11.1] -- but may not be used in a table declaration as a dimension bound.

c. A small amount of language revision is required to add a dynamic upper array bound to the language. On the other hand, modifications to the implementation will be significant. First of all, dope vectors will be needed to access dynamic arrays. Secondly, since JOVIAL does not have recursive procedures, the implementation need not allocate storage dynamically at procedure entry. Consequently, adding dynamic arrays may require the addition of a dynamic storage allocator to an implementation.

7. Records (A7).

a. Degree of compliance: P

b. JOVIAL permits entries of specified (i.e., machine dependent) tables to have alternative structures. This is accomplished by specifying the starting bit location of an entry. No discrimination condition is required. JOVIAL also has an OVERLAY facility that allows variables of arbitrary type to be overlayed without any discriminating condition [14, Section 1.5.2], thus circumventing type checking. In addition JOVIAL has a based allocation capability that allows arbitrary templates to be laid over any area in memory.

c. Hierarchically structured data can be achieved through the use of BLOCKs [14, Section 2.1.6]. However, all the component names must be unique since there is no name qualification and the only available operation on blocks is parameter passing.

d. Major modifications would be required to redesign the data structuring mechanism to allow suitable hierarchical structuring, and alternative record structures with discrimination. This is the case because such facilities are different in philosophy from the current data structuring facilities in the language and because of the interaction of such facilities with such features as assignment, parameter passing, and type checking.

AD-A037 639 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
CANDIDATE LANGUAGES EVALUATION REPORT.(U)
JAN 77 B M BROSGOL, R E HARTMAN, J R NESTOR DAHC26-76-C-0006
UNCLASSIFIED IR-217 USACSC-AT-76-11 NL

3 OF 6
ADA037639

Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. JOVIAL has no provision for encapsulated type definitions; the user has no means for defining assignment and access operations. The JOVIAL assignment statement can be used only for built-in types; it is not permitted to assign one table to another or one block to another. (This restriction is not explicit in [14], but it is implied by the syntax for <u>assignment:statement</u> [14, Section 3.3.1], since the right side must be a <u>formula</u> (thus of built-in type [14, Section 4.1]), and the rule which states that the value type of the <u>formula</u> must match the type of the variable(s) on the left side (non-built-in types never match under the rules of [14, Section 1.7]). Many conversions required to perform assignment are invoked implicitly [14, Section 1.7.1.2]. Reference to a table entry will retrieve the entry interpreted as a bit string. Since JOVIAL allows data overlaying, reference to a variable may retrieve a value different from the one most recently assigned.

c. In order to comply with this requirement (in the absence of encapsulated type definitions) assignment would have to be defined for tables and blocks. This, however, would require a substantial change to the language definition, as well as the implementation, since table and block declarations currently do not define new types. A fair amount of design is required to determine all the implications of treating tables and blocks as new types, and also to define the semantics of assignment.

d. Another change required is to remove the overlay facility so that data reference will always retrieve the last assigned value. This change is relatively minor. However, to fully ensure that data reference will always retrieve the last assigned value, based data (i.e., pointers) and implicit overlays in specified tables would also have to be eliminated, but such facilities are required by D6 and J4.

2. Equivalence (B2).

a. Degree of compliance: P

b. JOVIAL provides built-in operations which can be used to compare any two items. These operations cannot be used for comparison of tables or blocks, however. When the types of the data objects do not match, or their sizes differ, JOVIAL may implicitly convert one of the objects before comparing [14, Section 4.6.2].

c. For floating point identity to be specified within a precision, the user must explicitly specify it himself (e.g., ABS(A-B)<EPSILON rather than A=B).

d. Substantial revisions are required to satisfy this requirement. In order to define comparison for tables and blocks, they must first be redesigned as types of the language as discussed in B1. In addition, the implicit conversions need to be deleted so that objects of different types cannot be considered equivalent, and comparison for floating point needs to be redefined to include a precision specification.

3. Relationals (B3).

a. Degree of compliance: PT

b. JOVIAL defines relational operations for numeric data and for all types defined by enumeration [14, Section 5.2.2]. It is not possible to inhibit ordering, nor is it possible to define unordered sets.

c. The required modifications to the language are non-trivial. A new, unordered STATUS type, without mapping onto the integers, would have to be added to the language. Literals for this type would have to be differentiated from ordered STATUS literals.

d. The effect on the implementation would also be non-trivial. Determining the equivalence of two unordered STATUS types would require a great deal of checking. For example, the compiler would have to determine that the following three types are all equivalent: ("A", "B", "C"), ("B", "C", "A"), and ("A", "C", "B").

4. Arithmetic Operations (B4).

a. Degree of compliance: PT

b. The required operations are provided by JOVIAL, except that integer division generates an integer result. JOVIAL offers a modulo operator "\" [14, Section 5.2.2].

c. A fairly minor modification is required to add a division operation for integers which returns a real result.

5. Truncation and Rounding (B5).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement, since truncation only occurs when the result of an operation is outside the range specifications of the program. As stated in [14, Section 1.7.2] with regard to bit strings and integers: "If the size of the value exceeds the maximum size permitted for its type by an implementation, its size is adjusted. ...Leading bits are truncated." However, this truncation is implicit (no exception condition is signalled). With regard to floating point, however, JOVIAL allows the user to specify whether truncation or rounding is to be carried out (on the least significant bits) with the <u>truncation:option</u> [14, Section 2.1.3.2].

c. In order to completely satisfy this requirement, truncation of integers would have to generate an exception condition. This, however, would require the addition of an exception handling facility to the language (see our discussion of G7).

6. Boolean Operations (B6).

a. Degree of compliance: PT

b. JOVIAL provides the built-in Boolean operations AND, OR, NOT, XOR, and EQV. AND and OR may be evaluated in short circuit mode (and some implementations do so evaluate them) but [14] does not specify that short circuit mode must be used.

c. Fairly minor modifications are required to add the NOR operator and to require short circuit mode.

7. Scalar Operations (B7).

a. Degree of compliance: F

b. JOVIAL does not permit scalar operations or assignment on conformable arrays (tables) or records (blocks).

c. The modifications required are substantial. As discussed in B1, tables and blocks would first have to be redesigned as types of the language. Then rules for type equivalence would have to be specified and the definitions of applicable operations would have to be extended.

8. Type Conversion (B8).

a. Degree of compliance: P

b. JOVIAL provides explicit conversion functions among integer, floating point, bit string, and character string data [14, Section 1.7]. JOVIAL does not provide conversion operations between the object representation of numbers and their representations as characters. It is important to note that many of the JOVIAL conversion operations are actually a different interpretation of an unchanged bit pattern (e.g., the conversion from floating point to character interprets a floating point number as "n" characters right-adjusted [14, Section 1.7.1.2]). JOVIAL provides implicit type conversions for type incompatibilities in expressions, assignments, and parameter passing [14, Sections 1.7.1.2, 2.2.3, and 3.3.2]; the built-in types are considered to be a hierarchy (float, integer, bit-string, character-string, in high to low order), and implicit conversions can be applied from one type to any type higher in the hierarchy. However, because the only conversions that change an internal representation are those from integer to floating point, very little run-time overhead is likely to be incurred.

c. As discussed in A1, implicit conversions permeate the language to such a large extent that to remove them would require a major effort and substantially change the nature of the language.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: P

b. The only ranges permitted in JOVIAL are those determined from the number of bits declared for integer or float items. Explicit conversions are not required from one range to another. However, there is no run-time exception on truncation.

c. To comply with this requirement, a substantial modification would be needed to add an exception handling capability to the language (see our discussion in connection with G7).

10. I/O Operations (B10).

a. Degree of compliance: P

b. The JOVIAL language specification provides no operations allowing programs to interact with files, channels or devices. All such operations must be supplied in a procedure library; the definition of such a library is outside the scope of [14].

c. A large effort would be required to design a suitable set of I/O operations that fit cleanly into the language, provide the necessary capabilities, and are capable of interfacing with any existing operating system/file system without being implementation dependent.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. Although JOVIAL does allow enumeration types, it does not offer power sets. It does offer bit strings and all the logical operators described under B6 which are sufficient to define union, intersection, difference and complement operations.

c. A moderate amount of work would be required to add power sets of enumeration types directly into the language. This would involve defining the form that power sets should take, checking for interactions with other features, especially enumeration types, and defining the appropriate operations on power sets.

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: F

b. As stated in [14, Section 4.8.3], "the evaluation order of formulas is unspecified..." Thus, side effects need not occur in left-to-right order.

c. The change to the language to require left-to-right evaluation is very minor. A moderate amount of revision would be necessary to an implementation if it uses a different order of evaluation. This would also have a major effect on optimization (see our discussion in connection with J2).

2. Operand Structure (C2).

a. Degree of compliance: PT

b. JOVIAL has ten precedence levels, but some of these levels do not generally appear in a precedence table (e.g., indexing) [14, Section 4.8.3]. JOVIAL has some exceptions to standard combining rules: for example, the exponentiation operator, "**", combines left-to-right instead of right-to-left. Another uncommon feature of JOVIAL's expression definition is that two adjacent operators are allowed (e.g., A*-B is valid) [14, Section 4.8].

c. A fairly minor change would be required to make exponentiation right associative.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. JOVIAL satisfies this requirement. Variables but not constants are allowed as the targets in assignment statements [14, Section 3.3.1] and as the output arguments of procedure calls [14, Section 3.10.1]. Anywhere else a variable reference is allowed an expression is also allowed.

4. Constant Expressions (C4).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement. There is a facility for using constant expressions (called numbers, which is not an especially intuitive term for this concept) in some contexts, such as preset lists and size:specifiers in item:descriptions [14, Section 2.1.3.1]. It is implicit that evaluation be at compile-time, at least in some cases.

c. However, there are many contexts in which constants are required and where literal values (not expressions) must be used. Examples of this situation are table bounds [14, Section 2.1.5.3] and switch:point:index:group [14, Section 3.8.1].

d. A moderate amount of revision would be required to the language to locate and change all the places where constants but not constant expressions are currently allowed.

e. The change to the implementation would be fairly minor, since the facility to evaluate constant expressions at compile time currently exists.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: F

b. JOVIAL fails to satisfy this requirement, the main language deficiency being the inconsistencies in the rules for parameter passing to procedures [14, Section 2.2.3]. For example, tables and blocks may be input parameters but not output parameters; item parameters may be preset or overlaid but table and block formal parameters may not. (It might also be pointed out here that the parameter rules for procedures differ from those for "DEFINE"s, thus violating C5. Because of the different nature of these two language facilities, however, such variances are not surprising and probably not detrimental to language understandability.)

c. A very large effort would be required to satisfy this requirement. The parameter mechanism would have to be completely redesigned to make the rules consistent and this would substantially change the nature of the language.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement. Type checking is carried out for item parameters, but implicit conversions may be invoked. Type checking is not carried out for table or block parameters [14, Section 2.2.3]: "The attributes of the formal and actual parameters should match. However, no such restriction is enforced. If the attributes do not match, the programmer is responsible for ensuring that the location(s) he wishes to use for the formal parameter are those he specified."

c. As has already been discussed under A1 and B8, deleting implicit conversions and imposing strong type checking would require a large effort.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. JOVIAL distinguishes in the argument list and formal parameter list between input parameters and output parameters [14, Sections 2.2.1.1, 2.2.2.1, 3.10.1, and 4.10.1]. It should be noted, however, that nowhere in [14] is there a prohibition against assigning to an input parameter. This will have different effects depending on whether the parameter is an item or an aggregate: the former is passed by value, and the latter by reference.

c. Functions may only take input parameters [14, Section 4.10.1].

d. In addition to the data parameters provided, JOVIAL allows control parameters: statement labels, and procedure or function names [14, Section 2.2.1.1].

e. Modifying the procedure mechanism to correspond exactly to the four classes in this requirement would be a large effort. In addition to large revisions to provide the required form of constant and variable parameters (classes one and two), an exception handling control parameter mechanism (class three) would have to be designed.

8. Formal Parameters (C8).

a. Degree of compliance: F

b. JOVIAL fails to satisfy this requirement. The JOVIAL user must specify the types of all formal parameters (except for "DEFINE"s). JOVIAL does not support the kind of generic facility envisioned in C8.

c. A large effort would be required to meet this requirement. It should also be noted that the requirement is inconsistent since the macro capability implied by the phrase "instantiated at compile time by the characteristics of their actual parameters" does not provide a full "generic" capability.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. As stated in [14, Section 2.2.3]: "Actual parameters must match in position, number, and type with the called procedure's formal parameters..."

c. The modifications required to support this requirement would be non-trivial. The parameter passing mechanism would have to changed to meet the requirement, possibly by augmenting it with an implicit conversion from the variable size list of parameters on the calling side to a variable size array which is what the procedure itself would receive.

Section V. VARIABLES, LITERALS, AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: P.

b. JOVIAL partially satisfies this requirement: the DEFINE facility can be used to associate constant values (for item data) with identifiers. However, this facility is a general macro replacement mechanism which works by string replacement independent of context; thus there is nothing to prevent the programmer from using DEFINEd names for constants in contexts where identifiers are not permitted. Worse still, unexpected results may occur; for example, if K is DEFINED as "2 + 3", then the result of K * 5 is not 25, but rather 17 ("2 + 3 * 5").

c. Even with the DEFINE facility, there is no way to obtain identifiers which reference constant aggregates (i.e., tables or blocks).

d. A moderate amount of work would be required to add constant identifiers to the language. This effort would involve designing the form of declaration as well as appropriate rules and restrictions. In addition, suitable checking would have to be added to the implementation to prevent their use in writable contexts.

2. Numeric Literals (D2).

a. Degree of compliance: P

b. JOVIAL provides a syntax and a consistent interpretation for constants of built-in data types [14, Section 5.3]. However, [14] nowhere specifies I/O or the internal representations of external data.

c. The main effort involved here really involves adding I/O to the language (see our discussion in connection with B10). Once this is done, only a minor effort is required to ensure that numeric program literals and numeric data are consistent.

3. Initial Values of Variables (D3).

a. Degree of compliance: P

b. The JOVIAL user may specify the initialization (called "presetting") of statically allocated data only (static data is called RESERVE) [14, Section 2.1.7.2]. Dynamically allocated data may not be initialized. There are no default initial values. The reference document [14] does not mention that a variable should be defined before being referenced. RESERVE data which are not preset are considered "undefined."

c. It would be a minor change to the language to allow initialization of variables. However, a moderate amount of revision would be required to the implementation to support it and also to check for the use of uninitialized variables. Checking for uninitialized variables, moreover, can potentially require a large amount of run-time overhead.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. JOVIAL allows the user to specify the size, in bits, for integers (fixed point numbers are not in the language) and these integers may be signed or unsigned. The user may not restrict the range more narrowly than this size (e.g., with 4-bit unsigned integers it is not possible to restrict the range to Ø..9). High order bits are truncated when overflow occurs, with no exception generated. With regard to floating point numbers, the user can specify the size of the exponent (as well as the mantissa) in bits.

c. A moderate amount of effort would be required to modify the language and the implementation to require the specification of ranges on all variables and to provide appropriate checking. These modifications would change the nature of the language.

5. Variable Types (D5).

a. Degree of compliance: P

b. JOVIAL contains a variety of restrictions on the structure of data. The only records which can be array

(table) components are those comprising items; it is not possible to have a block as an array constituent. The number of dimensions in a table is restricted to 7.

c. A large amount of redesign would be required to remove the restrictions from JOVIAL to make it comply with this requirement. One aspect of this would involve making tables and blocks types in the language (see our discussion in connection with B1).

6. Pointer Variables (D6).

a. Degree of compliance: P

b. JOVIAL provides a pointer facility which can be used to build shared or recursively organized data structures; viz., the based alternative for allocation:specifier [14, Section 1.4]. However, declaring a data object as based means only that its data description will be used as a template for the area of storage referenced by the address formula; there is no user-controlled dynamic storage allocation in JOVIAL.

c. It is required in D6 that pointers be restricted to reference some declared type of data. JOVIAL completely fails to meet this provision; any integer variable or formula may be used as a pointer [14, Sections 1.4.1, 4.11.4, 3.10.1].

d. A large effort would be required to design a suitably restrictive and reliable pointer mechanism into JOVIAL to satisfy this requirement.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: P

b. Aside from its macro (DEFINE) facility, which is not directly relevant to E1, and its procedure and function mechanism, JOVIAL is quite weak in the area of definitional capabilities. Specifically, there is essentially no capacity for user-defined data types. (The status:list:name [14, Section 2.4.2] is the closest which JOVIAL comes to user-defined types, but this feature lacks the security generally associated with data types. For example, the relationship between status and integer values is explicit, and status constants are usable as integers and vice versa.) JOVIAL's data facility is that of "data structuring", as found also in FORTRAN, COBOL, TACPOL, and PL/I.

c. There is no facility in JOVIAL for defining new prefix or infix operators nor for extending the meaning of existing operators.

d. An extremely major design effort would be required to add abstract data definition and operator extension facilities to JOVIAL. Such facilities should be incorporated into a language right from the start, during its initial design. Adding such facilities after the fact would be extremely difficult, if not impossible, because of the pervasive effects of such facilities. Interactions of such facilities with type checking, assignment and parameter passing would require a large design effort.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. JOVIAL does not provide defined types and therefore fails to meet this requirement.

c. Meeting this requirement is clearly contingent on adding user-defined types to the language. Even so, it is not yet within the state of the art in language design to make user-defined types indistinguishable from built-in types.

3. No Default Declarations (E3).

a. Degree of compliance: T

b. There is no provision for implicit declarations in JOVIAL.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. JOVIAL provides neither type definition nor operator extension facilities.

c. A major design effort would be required to add an operator definition capability to the language as has already been discussed under E1.

5. Type Definitions (E5).

a. Degree of compliance: F

b. JOVIAL has no facilities for defining new types.

c. As already mentioned under E1, a major effort would be required to add type definitions to JOVIAL.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: P

b. JOVIAL partially meets this requirement. Data may be defined by enumeration (STATUS values) and as Cartesian products, but not by discriminated union nor as the power set of an enumeration type. The definitions are processed entirely at compile time.

c. As has already been discussed under A7, major modifications would be required to add discriminated union to JOVIAL. As has been discussed under B11, moderate revisions would be required to add power sets.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The OVERLAY facility, based variables, and implicit overlays in specified tables all provide free union. JOVIAL provides no type subsetting.

c. The OVERLAY facility can be easily deleted from the language. Implicit overlays in specified tables are part of the machine-dependent object representation specifications required by J4, and based variables are part of the pointer mechanism required by D6.

8. Type Initialization (E8).

a. Degree of compliance: F

b. Lacking type definition, JOVIAL has no facilities for meeting this requirement.

c. Meeting this requirement is highly dependent on adding type definition facilities to the language, which is a major change (see our discussions of E1 and E5.).

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

a. Degree of compliance: PT

b. JOVIAL satisfies this requirement fairly well. Data may be permanent, temporary, or based. Permanent (RESERVE) data have the behavior of "own" variables in ALGOL; their allocation scope exceeds their access scope. For temporary (IN) variables, the two scopes are identical. However, because of the absence of checks for based data, this facility allows unlimited access scope to data which may not be allocated.

c. The possible abuse of based data stems from the fact that no allocation scope actually exists for this form of variable. A based data structure simply defines a template which may be laid upon any region of memory and does not require allocation. Limitations on the range of overlaying are needed. These are difficult to define while still preserving based data's essential properties. However, these properties will be lost in any case if strong type-checking is required. One method of enforcing the rules of F1, then, would be to require the programmer to explicitly define the memory regions (variables) and possibly locations within these regions upon which the based data template may be applied. This would necessitate a moderate amount of language redesign and compiler modification. Some run-time checking would be required to assure that the base formula has the appropriate value.

2. Limiting Access Scope (F2).

a. Degree of compliance: P

b. JOVIAL partially satisfies this requirement. In addition to the scope rules mentioned in the preceding paragraph, the DEF and REF facility allows access to be limited both where an entity is defined and where it is used. However, there is no facility for renaming to prevent name conflicts.

c. A renaming facility must be added to the REF statement to bring JOVIAL into full compliance with F2. Only minor language modifications, which do not impact other

language features, are required to accomplish this. The additional compiler support needed would be small.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: T

b. The scope of JOVIAL identifiers is wholly determined at compile-time [14, Section 1.3.3]. JOVIAL access scopes are lexically embedded with the most local definition applying when the same identifier appears at several levels.

4. Libraries Available (F4).

a. Degree of compliance: PT

b. JOVIAL has only a small library of specified intrinsic functions [14, Section 4.10.3]. The JOVIAL COMPOOL feature allows easy sharing of source language library modules and facilitates the devlopment of such libraries. However, [14] does not specify any application-oriented data or operation libraries.

c. The development of extensive application libraries is outside the scope of language definition. To the extent that such libraries can be encouraged by language design, through mechanisms which provide safe and easy accessibility, JOVIAL complies with this requirement (see our discussions in connection with F5 and F6). The one limitation on ease of access rests in JOVIAL's failure to provide renaming capabilities; however, support of such facilities can be simply added (see our comments concerning requirement F2).

5. Library Contents (F5).

a. Degree of compliance: T

b. JOVIAL offers the COMPOOL facility, which satisfies this requirement. The !LINKAGE directive [14, Section 6.2] allows access to procedures written in other languages.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: T

b. The JOVIAL COMPOOL allows the definition of any JOVIAL data structure or routine to be included in a compile-time accessible form [14, Section 2.6.3.1]. The user can select any data objects or routines from the named COMPOOL. JOVIAL offers also a COPY directive, which causes external source text to be included in the to-be-compiled text. When refererencing a COMPOOL-defined identifier, the program obtains also all the attributes of that identifier [14, Section 2.6.4.2].

7. Standard Library Definitions (F7).

a. Degree of compliance: F

b. The JOVIAL specification [14] does not define interfaces to operating system capabilities, peripheral equipment, or any other machine dependent capabilities. Such capabilities must be supplied in a library but the contents and use of the library are implementation dependent and not specified in [14].

c. The most glaring deficiency in this area is that [14] does not define input or output operations. To correct this flaw alone would require extensive additions to the language definition. While all compilers must support I/O in some manner, a common description could be difficult to implement on a given machine. Modification of existing compilers to provide these new constructs would require a large effort (see our comments in connection with B10 for further discussion of I/O).

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structrures (G1).

a. Degree of compliance: P

b. JOVIAL provides control structures for sequential, conditional and iterative control. Using "based procedures" [14, Section 1.4.2] it is possible to produce a limited recursive control mechanism. No control structures are available in JOVIAL for parallel processing (pseudo or real), exception handling or asynchronous interrupt handling. The control structures of JOVIAL are composable.

c. JOVIAL's deficiencies in this area cannot be easily removed. The addition of the required facilities would entail major language revisions. Compiler managed recursion requires new (dynamic) methods of memory allocation (see our discussion in connection with G5). Parallel processing necessitates provisions for task creation, termination, and synchronization. In addition, controlled access variables or program regions must be introduced. This would be complicated by the multiple reference methods supported by JOVIAL (see our comments regarding requirement G6). Compiler support of recursion and parallel processing needs to be extensive.

d. Exception and asynchronous interrupt handling, while somewhat simpler to define and implement, constitute far from trivial additions. Exception situations must be explicitly defined throughout [14]. In addition, general treatment of such instances requires a much more detailed model of the JOVIAL-machine interface.

2. The GoTo (G2).

a. Degree of compliance: P

b. JOVIAL provides a GoTo operation with very few restrictions on its use. Not only may a GoTo jump to any labelled statement in the most local scope of definition or in the enclosing program module [14, Section 3.4.1] but a procedure may also have statement label input parameters and GoTo those labels [14, Section 2.2.3]. JOVIAL does not have label variables or numeric labels. The language does prohibit branching into loops [14, Section 3.9.2], in that

the result of such an operation is "undefined." In addition, the REF declarator [14, Section 2.6.2], permits GoTo operations which greatly abuse scoping rules. For example, using such a declaration, a jump may be executed into the middle of a procedure which is outside the scope of the GoTo Statement, or from within a REFed routine into a textual ancestor (*) which has not been activated.

c. Restricting the scope of control transfers would be relatively simple. However, in the context of such restrictions, label parameters must be removed from the language. The major drawback of this is that in the absence of exception handling facilities (see our discussions regarding G1 and G7) these label parameters provide a reasonably clean method of passing error recovery information.

3. Conditional Control (G3).

a. Degree of compliance: PT

b. The JOVIAL IF-statement is not fully partitioned in that the ELSE clause is optional [14, Section 3.7.1]. JOVIAL's IF-statement has an unusual syntax: IF condition; statement-executed-if-true; [ELSE clause;]. The word THEN does not appear.

c. JOVIAL also offers a case statement (called a SWITCH [14, Section 3.8]). The selection condition is an integer expression, and the cases are labelled with the integer values which cause selection. There is a DEFAULT case, which is selected when no other case matches. The use of comma vs. blank to govern "fall-through" behavior is error-prone.

d. Requiring conditionals to be fully partitioned necessitates only small language and compiler modifications. The ELSE and DEFAULT clauses may no longer be considered optional. The null statement is already available for use in situations where no action is desired.

----

* A textual ancestor of a procedure, P, is considered to be any other procedure in which P has been embedded (nested).

4. Iterative Control (G4).

a. Degree of compliance: P

b. The iterative control structure of JOVIAL permits the termination condition to appear only at the beginning of the loop [14, Section 3.9.1]. JOVIAL does not impose unnecessary overhead for common special case termination conditions. The JOVIAL loop control variable is global to the loop; [14] does not specify any meaning for loop control variables after loop termination.

c. Since [14] does not provide a specific definition of the loop control variable's value upon termination, the addition of a statement to the effect that it is unavailable would be trivial. However, while this does preclude implementation-dependent interpretations, it is not quite the same as a local definition due to a greater allocation scope. Freedom of placement for the termination test requires the addition of a new control statement. A general form of loop-while should replace the JOVIAL while statement. This involves minor changes to the language definition and compiler design. Probably the greatest effort would be that of agreeing on a suitable form for this statement.

5. Routines (G5).

a. Degree of compliance: P

b. Recursion is not explicitly available in JOVIAL [14]. (A recursive:directive, with neither syntax nor semantics specified, is listed [14, Section 6.0]. This directive is ignored.) The JOVIAL user does have the possibility to use "based procedures" to implement recursive routines. As this approach is not supported directly by the language, it is error prone and clumsy to use.

c. Some form of compiler-supported recursion was, obviously, meant to be included in the full JOVIAL/J73 language. The level I subset, however, does not define what form this was meant to take. The most likely reason for this rests in a desire to support only those facilities which do not require dynamic memory llocation. J73/I does not contain any such facilities and for this reason implementation would be complicated by the addition of

recursion. Variables declared to be IN (allocated upon procedure entry) within recursive procedures need to be allocated from a memory pool (commonly called a heap). Routines must be included in the run-time system for maintenance of this memory pool and references to heap variables.

d. No change to the language definition, [14], is required to support recursive procedures; however, the meaning of the !RECURSIVE directive should be defined. The unrestricted nature of the JOVIAL GoTo may pose problems. If a statement label within a recursive procedure is made available via a REF declaration sticky situations can arise. Consider, for example, a recursive procedure P. P already has several existing activation records (i.e., it has been called but not exited several times). A GoTo causes transfer of control to a label internal to P from some external point. By the rules of [14, Section 2.6.2] no IN variables can be accessed and a RETURN would be illegal if the GoTo was reached by way of a procedure call, but legal if it was reached by another GoTo. Such rules are difficult to understand and enforce.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. JOVIAL contains no facilities for meeting the parallel processing requirement.

c. The addition of parallel processing facilities to JOVIAL would constitute a major task. The effort would be more difficult than the complete definition of such facilities would have been if done along with initial language design. The reason for this is that interaction with existing language features greatly complicates the required changes. For example, one needs the capability to prohibit simultaneous access and update of a variable by two parallel processes. In the present language this is made very difficult by the existence of multiple methods for addressing a storage location (e.g., by name, by overlayed name, or by based formula).

7. Exception Handling (G7).

a. Degree ofcompliance: P

b. JOVIAL fails to satisfy this requirement. The issue of exceptions -- either hardware-oriented (e.g., overflow) or software-detected (e.g., index out-of-bounds) -- is not addressed very deeply in [14]. The explicit conversion functions allow as an exception-handling argument a statement label to which an unconditional branch is made when the conversion causes a value change [14, Section 1.7.2].

c. I/O, and therefore I/O exceptions, are not defined in [14].

d. In terms of language definition alone this requirement necessitates extensive modifications. The issues of error detection are for the most part ignored in [14]. All possible exception conditions must be identified and a mechanism for recovery must be introduced. The lack of defined I/O facilities greatly complicates this task (see B1Ø for problems associated with their introduction). The difficulties encountered in implementation will depend upon the sophistication of the added facilities and the machine on which the programs are to run. Even with minimal facilities and a high degree of machine support this would require quite large additions to the compiler.

8. Synchronization and Real Time (G8).

a. Degree of compliance: F

b. JOVIAL provides no source language features for synchronization of parallel processes, nor for access to hardware interrupts or real time clocks.

c. Let us first mention that this requirement assumes that both requirements G6 and G7 have been met. The fact that JOVIAL fails to comply with either of these implies that a great effort is needed before G8 can even be considered. Given compliance with both G6 and G7, the facilities suggested here still constitute a major change to the language. The ability to specify relative priorities and permit delay on any control path necessitate sophisticated sequencing of parallel processes. To handle

asynchronous hardware interrupts one must be able to save the state of the present process and resume it at some future time.

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. The JOVIAL language is free format with an explicit statement delimiter, allows the use of mnemonically significant identifiers, has a relatively easily parsed grammar, and does not permit abbreviation of identifiers or keywords. Although JOVIAL keywords may not be abbreviated some of the mnemonics are already so abbreviated that they are not easily read. The single character mnemonics used in declarations (F=float, U=unsigned integer, D=densely packed) are examples. Also, some mnemonics are poorly chosen (e.g. IN does not specify integer type but rather automatic allocation of storage upon entry (in?) to a scope). The absence of the word THEN in the IF-THEN-ELSE construct is unusual. In addition, table declarations contain many context-dependent symbols and values which can obscure readability. For example, the following is a valid specified table declaration:

TABLE T1, [3] P 5 F 5, 3 D [2,1]=[0] 2 (5) ;

c. It should also be noted that the JOVIAL grammar given in [14] is ambiguous, as revealed in the rule in [14, Section 4.8.1]:

<u>numeric:formula</u> ::=
<u>numeric:formula</u> <u>arithmetic:operator</u> <u>numeric:formula</u>

The ambiguity is resolved in the rules for operator precedence.

d. As can be seen from the above example, JOVIAL's declarations can be hieroglyphic to those not intimately familiar with the language. The introduction of more significant and readable declarators and possibly guidewords would greatly ease the task of understanding JOVIAL programs. This necessitates no change in language semantics. The enhanced syntax would require only minor modifications to the scanning and possibly parsing modules of the compiler.

e. The ambiguity of JOVIAL's grammar can be justified as a means of facilitating readability of the defining document. Implementors are not restricted to use of the grammar in its specified for and in fact would probably wish to avoid the specified ambiguities. This could, however, lead to somewhat implementation-dependent syntax unless great care is exercised. We suggest that where it is felt necessary to use ambiguous productions, an unambiguous set of rules should also be included for implementation purposes. This constitutes a fairly minor addition to [14].

2. No Syntax Extensions (H2).

a. Degree of compliance: T.

b. JOVIAL has no facilities for modifying the source language syntax (the DEFINE feature is not regarded as violating this requirement).

3. Source Character Set (H3)

a. Degree of compliance: T

b. The JOVIAL source character set consists of the upper case letters A-Z, the digits 0-9, and the following special characters [14, Section 5.1]:
blank + - * / \ < > = : , ; () [ ] ' " ! $
All of these characters are in the 64-character ASCII subset.

4. Identifiers and Literals (H4).

a. Degree of compliance: P

b. The JOVIAL language definition provides formation rules for identifiers [14, Section 1.3.1] and literals [14, Section 5.3.3.1]. The prime (i.e., apostrophe) is used for an explicit break character in identifiers [14, Section 1.3.1]. No break character is defined for use within literals. End of line has no role in JOVIAL.

c. Full compliance with H4 can be reached with only minor changes to the language and scanning module of the compiler. One must permit a break character within literals (e.g., the apostrophe) and require separate quoting of multi-line character literals.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. JOVIAL nowhere specifies any function for end-of-line. Any implementation-defined "other characters" are accepted in character string constants and comments [14, Section 5.1].

c. The requirements of H5 can be met with minor additions to [14]. An explicit end-of-line constant for use within character literals must be defined. A prohibition of the continuation of a lexical unit across lines must be added. Both of these involve trivial changes to the compiler and do not impact other language features.

6. Keywords (H6).

a. Degree of compliance: P

b. JOVIAL's keywords are reserved and are 68 in number [14, Section 5.2.1]. These keywords are generally informative, although DEF, DEFINE, and DEFAULT are not completely clear. Some keywords, particularly the implementation constants describing the target machine, are intended to appear anywhere an integer constant or identifier can appear.

c. The freedom with which implementation constants can be used cannot be restricted without limiting their value. Possibly their definition as keywords is ill-advised, but allowing programmers to declare variables with these names would only lead to obscurity and confusion. The unclear keywords can be simply changed to more informative ones (e.g., DEFAULT which connotes a switch path to be taken when all others fail might be replaced by OTHERWISE). This would constitute a trivial change to the compiler's scanner.

7. Comment Convention (H7).

a. Degree of compliance: PT

b. JOVIAL comments are enclosed in quotation marks [14, Section 5.4] and may contain any character except a quotation mark. End-of-line has no effect on comments. JOVIAL comments may appear almost anywhere in programs. An

important exception is in connection with macros (DEFINE): since the define:string is delimited by quotation marks, comments may not appear between the define:name and the define:string. This, however, is a good place for the user to provide a comment explaining the define:declaration.

c. The JOVIAL comment convention may be trivially modified to accept end-of-line as a terminator. However, this implies that it could not be used to enclose a program region of arbitrary length and may be construed as a violation of the balanced parentheses requirement. The conflict between comment and DEFINE notations can be removed by defining a different comment delimiter. Use of this facility (i.e., commenting within DEFINEs) could lead to subtle errors due to the macro replacement of a DEFINEd name in a place where comments are not permitted. All in all, full compliance may be reached with great ease.

8. Unmatched Parenthese (H8).

a. Degree of compliance: T

b. JOVIAL has parentheses "()", brackets "[ ]" and BEGIN END as "parentheses". These must all be matched.

9. Uniform Referent Notation (H9).

a. Degree of compliance: F

b. The argument list for a function call is enclosed in parentheses "()" [14, Section 4.10.1]; the subscript list for a table reference is enclosed in brackets "[ ]" [14, Section 4.11.1].

c. JOVIAL uses pseudo-functions for its bit- and character substring operations [14, Section 4.11.2]: these pseudo-functions may appear on the left of an assignment statement.

d. Within the program body, no real difficulty arises from permitting parentheses to be used for both function and table references; however, ambiguities will be encountered if this is allowed within table declarations. A,[2]B and A,(2)B both have unique and valid meaning within a specified:table:declaration preset:part. If we limit our requirement to uniform reference within the program body

then compliance will necessitate a moderate amount of effort, otherwise a large amount is required.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. JOVIAL has several inconsistencies: the equal sign "=" may be the assignment operator or a relational operator (e.g., IF XX=1; XX=2; ELSE XX=5; [14, Section 3.7.3]), the letter "M" in a table declaration specifies a medium packing density [14, Section 2.1.5.5] and in a floating point constant literal specifies the mantissa precision [14, Section 5.3.3]. The representation of status constants (the JOVIAL enumeration type) is V(name) which looks like a function call but is in fact an integer constant [14, Section 5.3.3.1]. The single quote may be used in identifiers as a break character [14, Section 1.3] but is also used to delimit bit and character string constants [14, Sections 5.3.4 and 5.3.5]. As mentioned in the preceding paragraph, the body of a define:declaration is delimited by double quotes, which are also used to delimit comments. The parameters in a define:declaration are preceded in the define:string by an exclamation mark, which also precedes all compiler directives [14, Section 6].

c. The overall notational change needed would require a moderate amount of language redesign. The modifications are all simple syntactic changes and therefore would require only small revisions to the compiler.

Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

   a. Degree of compliance: P

   b. The behavior of any program in the event of run-time exceptions, compile-time errors or situations yielding undefined results (e.g., [14, Section 3.9.2]) are not specified and thus are entirely implementation dependent. Within a program all attributes must be declared or a JOVIAL specified default is selected (e.g., precision for floating point variables).

   c. The task of eradicating all defaults from JOVIAL would not be easy. Many implementation-dependent defaults exist and permeate the entire language. Error situations are almost totally ignored in [14]. The definition provides the syntax and semantics of valid JOVIAL statements but supplies little information about actions to be taken when encountering invalid code. No error or warning messages are defined. The needed modifications to the defining document are extensive. Such changes would at least require moderate revision of the compiler.

2. Object Representation Specifications Optional (I2).

   a. Degree of compliance: P

   b. JOVIAL provides a very high degree of user control over data representation, but there are well-defined defaults when the user does not care. There is no user control over program characteristics such as open vs. closed subroutines, and reentrant vs. non-reentrant code generation.

   c. Control over program characteristics such as open vs. closed routines and reentrant vs. non-reentrant code generation can be provided through the addition of new directives to the language. These additions could be easily made without seriously impacting any other language features. The needed modifications to the compiler would entail a moderate amount of work. Compilation of reentrant code may involve penalties in run-time efficiency.

3. Compile Time Variables (I3).

a. Degree of compliance: T

b. JOVIAL provides a set of compile-time constants, called machine:parameters [14, Section 1.2], which specify characteristics of the object machine. In addition, the DEFINE facility, COMPOOL, and certain compiler directives (such as !SKIP) provide a flexible compile-time capability to the user.

4. Conditional Compilation (I4).

a. Degree of compliance: T

b. JOVIAL satisfies this requirement, via its skip:directive [14, Section 6.1.2]. However, it would have been a somewhat more unified approach if the conditional compilation facility were part of the DEFINE mechanism.

5. Simple Base Language (I5).

a. Degree of compliance: P

b. JOVIAL satisfies this requirement fairly well. Aside from the DEFINE feature, JOVIAL lacks extension facilities; thus the entire language may be considered as the base. Complexities in the language include a somewhat baroque switch:statement and interactions between features such as storage allocation, presetting, and procedure parameters.

c. The extent of modifications required to achieve full compliance with this requirement greatly depends upon the additions made in conjunction with other Tinman requirements. For example, if class structures as discussed in section E are desired, some facility must be included within the "base" for their definition.

d. As the language stands, the modifications necessary for the definition of a simple base are relatively minor. A few features, such as the switch, could be removed and defined in terms of other constructs, but the result would not differ greatly from the total language.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. JOVIAL limits the number of array dimensions [14, Section 2.1.5.3.2] and the length of identifiers [14, Section 1.3.2]. Neither the number of nested parentheses levels in an expression nor the number of identifiers are restricted by [14].

c. Setting values for the various translator-dependent limits would not be a difficult task. The real problems rest in differentiating limits imposed by translator design and those inherent in the object machine. For example, if we limit the number of identifiers to 400, which seems to be in the range suggested, a translator on a 16-bit machine would require 6200 words to store all potential identifiers. This number is arrived at by assuming 2 character/word storage and noting that the first 31 characters of a JOVIAL name are significant. On a small machine this could be prohibitive.

7. Object Machine Restrictions (I7).

a. Degree of compliance: PT

b. No restrictions dependent only on the object environment are built into the JOVIAL language definition [14]. Nowhere in [14] is it stated that JOVIAL will report when a program exceeds the resources or capabilities of the intended object machine.

c. Reporting of resource overruns should be done by any "good" compiler and the inclusion of such a requirement within [14] is trivial. We should note that such inclusion runs counter to the nature of [14], in that in general error situations are not addressed.

Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. JOVIAL satisfies this requirement fairly well. The data structures offered by the language are machine-oriented (e.g., size and precision are specified in terms of numbers of bits, table entries are regarded as bit strings), so that the programmer is always aware of the match between data and hardware. Array bounds are fixed at compile-time, avoiding the need for dope-vectors. Recursion is possible only at great pains by the programmer; when recursion is achieved, the costs are explicit. Data overlaying is permitted (there is no discriminated union). The programmer can specify the layout of tables as parallel or serial [14, Section 2.1.5.4], and can indicate the density of packing. Optimization directives such as !INTERFERENCE and !REDUCIBLE, are available [14, Sections 6.4 and 6.5].

c. An example of a language feature which results in possibly-unexpected overhead is implicit conversion. A misguided efficiency is also illustrated by this feature: the conversion from floating-point to character-string simply reinterprets the bits comprising the former as though they were characters. Although this results in no overhead, the behavior is not likely to be the one desired.

d. Removal of the implicit conversions existing in JOVIAL is discussed in B8. As can be seen from that discussion the changes are far from trivial.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: F

b. JOVIAL does not satisfy this requirement. The order of evaluation of formula components [14, Section 4.8.3] and of arguments to a routine [14, Section 2.2.3] is left unspecified. Thus side effects need not occur in left-to-right order.

c. This requirement has the effect of greatly limiting the number of optimizations applicable. Recognition of all

conceiveable side-effects is frequently difficult and sometimes impossible. The JOVIAL compiler is permitted great freedom in terms of optimizing the user's program. Data structures may be rearranged and formulas evaluated in any order. Compliance with J2 would entail extensive compiler improvement or a potentially large penalty in run-time efficiency. These effects would be limited, however, to the optimization section of the compiler. The addition of this requirement to the language definition [14], however, would be trivial.

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. JOVIAL makes no provision for the insertion of machine language code. However, the user has access to machine capabilities at a very detailed level in the data representation. An OVERLAY specification can define an absolute memory address for the storage of data. By using JOVIAL's "specified table" capability, the programmer can associate an identifier with fields of any width within a word or words. The OVERLAY facility allows the same field to be used, for instance, as an unsigned integer and as a bit string. Thus, although these fields may be manipulated only through JOVIAL constructs, the effect is to allow quite detailed control of the hardware on certain types of computers.

c. The REF mechanism and !LINKAGE directive can be used to provide a machine language insertion capability. This directive is, of course, implementation-dependent. The encapsulation of these routines can be achieved via the conditional compilation of the associated REF.

d. One stumbling block still remains, however. Since JOVIAL provides no mechanism for defining open (in-line) procedures every reference to a machine language insertion necessitates a procedure call. This can be disastrous when these insertions are being used to achieve improved efficiency. The introduction of open procedures is discussed in J5 but further work might be required to support routines called through a !LINKAGE directive.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. JOVIAL offers comprehensive capabilities for the specification of data representations, as described above in conjunction with J1. However, the facilities provided are not encapsulated: they are specified together with the logical description of the data.

c. Separation of the logical definition and object representation description would be difficult in JOVIAL. The specified:table:declaration is basically an ordinary:table:declaration with added facilities for specifying item placement. A great amount of redundancy would be required for separate specification, and compilation would most likely become more difficult. The !SKIP, !BEGIN and !END directives allow any specified:table:declaration to be encapsulated in compile-time conditional statements.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: F

b. JOVIAL contains no facilities for specifying open vs. closed routine calls.

c. From a language definition viewpoint this requires only a trivial change. The introduction of an !OPEN directive would suffice. Implemenmtation, however, would not be so simple. Several features of JOVIAL would cause complications. For example, the meaning of a REFed statement label would be ambiguous if it appeared within an open procedure with several calls. In addition, the meaning of an open based procedure is not immediately obvious. The DSIZE function [14, Section 4.10.11] would have to be extended. In summary, for a coherent treatment of open and closed routine calls a moderate amount of language redesign and compiler revision would be needed.

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

   a. Degree of compliance: T

   b. The JOVIAL defining document, [14], makes no reference to the existence of an operating system. Language features, common to other languages, which rely on some OS support are not defined in JOVIAL. These include input and output, parallel processing and interrupt capabilities.

2. Program Assembly (K2).

   a. Degree of compliance: T

   b. JOVIAL encourages modularity through the !COPY and !COMPOOL directives and the DEF and REF statements.

3. Software Development Tools (K3).

   a. Degree of compliance: U

   b. The existence of support tools is not discussed in [14]. It should be noted that some of the recommended tools do exist for at least one implementation of the language.

4. Translator Options (K4).

   a. Degree of compliance: U

   b. Several options such as listing suppression are supported in JOVIAL through directives, but general compiler options are not discussed.

5. Assertions and Other Optional Specifications (K5).

   a. Degree of compliance: F

   b. Except for the standard comment mechanism JOVIAL does not provide any facility for the inclusion of assertions, assumptions, axiomatic definitions or units of measure.

   c. The introduction of additional commenting forms for these purposes is straightforward.

Section XIII. TRANSLATORS.

1. No Superset Implementations (L1).

a. Degree of compliance: F

b. JOVIAL J73/I is a subset language and therefore totally fails to meet this requirement. Several keywords and directives are defined simply for compatibility with the full language.

c. Removing the keywords and directives which are not defined in J73/I would be trivial. Outlawing superset implementations, however, is antithetical to the intentions of its designers for the language to be a subset of full J73.

2. No Subset Implementations (L2).

a. Degree of compliance: U

b. [14] does not consider the possibility of subset implementations. It would be trivial to add such a prohibition to the defining document.

3. Low-Cost Translation (L3).

a. Degree of compliance: U

b. The efficiency of compilation is outside the scope of language definition. To the extent the language design affects compiler speed, JOVIAL meets the requirement. No features of the language require excessive computation to determine their meaning.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This requirement applies only to language implementation.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. Cross-compilation is an issue of language implementation, not design.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. Compiler responsibilities are not discussed in [14].

7. Diagnostic Messages (L7).

a. Degree of compliance: F

b. [14] does not suggest a set of error and warning messages.

c. It would require a large effort to define a set of error and warning messages for JOVIAL. The major difficulty arises from the unclear nature of many possible error situations in [14]. Frequently, potential misuse of JOVIAL contsructs is ignored (see our discussion regarding M2).

8. Translator Internal Structure (L8).

a. Degree of compliance: T

b. [14] does not dictate the internal structure of a JOVIAL compiler.

9. Self-Implementable Language (L9).

a. Degree of compliance: T

b. The DAIS JOVIAL J73/I compiler is written in JOVIAL.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: T

b. No features of JOVIAL are beyond the current state-of-the-art.

2. Unambiguous Definition (M2).

a. Degree of compliance: F

b. The omissions and unclear specifications of the defining document, [14], are numerous. The semantics given for many language features are vague and open to several interpretations (see c below). Frequently the semantics of syntactically valid constructs are omitted or incomplete (see d, e and f below). Examples which are intended to illuminate sometimes achieve the opposite effect by being incorrect (see g and h below). Use of such words as "can", "may" and "should", without further explanation only lead to confusion (see i below). Occasionally, the consequences of the stated semantics seem to be unintended (see j below).

c. Type matching rules, one of the most importnt aspects of any language definition, are unclear for JOVIAL. The following are the first two sentences of [14, Section 1.7] pertaining to these rules, "Two values are said to match if their value types and sizes are identical. In contexts requiring a particular value type or an equivalence of value types of two operands, the value type of a value can be changed...". The document then goes on to describe implicit conversions. The question still remains, however, "What is the meaning of 'types must match'?" This is of crucial importance since the requirement about 'matching' types appears throughout [14].

d. The semantics of a FOR statement are incomplete. It is obvious from the syntax that the initial:value clause for a control:variable may be omitted; however the semantics of such an omission are not specified [14, Section 3.9].

e. In the description of the RETURN statement [14, Section 3.6.2], it is stated, "If a return is from a (lexically) outer procedure, any output parameters of the

inner procedure that contain the return:statement are not set." But what happens if the output parameter of an inner procedure happens to be associated with an output parameter of the exited procedure?

f. Probably the most glaring omission of [14] occurs in the definition of string padding [14, Section 4.6.2]. Bit strings are specified to be padded with zeros, but the character padding element is never defined.

g. The Shell sort on [14, Page 3-12], at best, sorts in reverse order.

h. The result of the logical formula given on [14, Page 4-3], is said to be 3B'16571' when, in fact, it should be 3B'16471'.

i. In [14, Section 1.7.1.2] the following statement is made, "If the type of two values is B, U, S or F, a matching function can be invoked implicitly..." Does the word "can" imply that these implicit conversions are implementation-dependent? If so, it should be stated explicitly.

j. According to the language semantics given in [14], the following statement is illegal if II is an integer (even unsigned):

```
II=II**II;
```

The reason it is illegal is that an exponentiation where the exponent is not a positive integer constant returns a floating point value [14, Section 4.8.2] and such a value may not be implicitly converted to integer type [14, Section 1.7].

k. It would be quite difficult to remove all ambiguous specifications from [14]. Large and numerous sections of the document would have to be rewritten.

3. Language Documentation Required (M3).

a. Degree of compliance: F

b. The language definition includes the syntax, semantics, and examples of each language construct. However, the syntax is frequently ambiguous (see our discussion in connection with H1) and the semantics unclear

(see our comments regarding M2). In addition, examples are sometimes incorrect (see our remarks concerning requirement M2).

c. The scope of modifications required to produce a clear and unambiguous language definition have already been seen to be substantial (see our final remark regarding M2).

4. Control Agent Required (M4).

a. Degree of compliance: U

b. The control of language and compiler standardization is outside the scope of language definition.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. Identification of "support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids" is beyond the scope of language definition. One should note that in reference to JOVIAL this identification process should be facilitated by the fact that support agents already exist.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is beyond the scope of language definition. The comments of M5 apply here.

Section XV. CONCLUSIONS REGARDING JOVIAL (J73/I)

1. Objectives of JOVIAL and Tinman.

The only goal of the Tinman that JOVIAL comes close to meeting is that of efficiency; JOVIAL does especially poorly with regard to reliability and readability. Even with regard to transportability, JOVIAL does not do very well since the language permits machine and implementation dependencies to permeate an entire program.

2. Summary of Major Areas of Conflict between JOVIAL and Tinman.

a. Data and Types. There are several major conflicts here. JOVIAL lacks strong typing and performs implicit conversions. In addition, it does not have a fixed point data type, user-defined character sets, dynamic arrays, nor discriminated variant records.

b. Operations. There are several major conflicts in this area. JOVIAL does not have assignment or comparison operations for tables or blocks, lacks I/O facilities entirely, and does not have a powerset type.

c. Expressions and Parameters. One conflict in this area is that JOVIAL does not require left-to-right evaluation. With regard to parameters, the rules are not consistent, implicit conversions are sometimes performed, the kinds of parameters do not correspond to those in the Tinman, there is no generic capability, and a variable number of parameters is not allowed.

d. Variables, Literals, and Constants. The major conflict in this area is the unreliable nature of the pointer mechanism. In addition, ranges cannot be specified on numeric variables, not all variables can be initialized, and there are various restrictions on the structure of data.

e. Definition Facilities. The major conflict is the lack of a type definition facility and an operator extension facility. In addition, there is no discriminated union or power set type in the language, and free unions (i.e., overlays) are permitted.

f. Scopes and Libraries. The major conflict in this area is the lack of a machine independent interface to operating system and I/O capabilities.

g. Control Structures. The major conflicts in this area are the lack of recursion, parallel processing and exception handling capabilities. In addition the language has an unrestricted GoTo statement.

h. Syntax and Comment Conventions. The conflicts in this area are the ambiguous grammar, the poor choice of keywords, and the lack of uniform referent notation.

i. Defaults, Conditional Compilation, and Language Restrictions. The major conflict in this area is the fact that often the behavior of a construct, particularly in the case of an error situation, is either unspecified or explicitly left undefined. In addition, the user does not have control over such characteristics as open vs. closed subroutines, and reentrant vs. non-reentrant code.

j. Efficient Object Representations and Machine Dependencies. The major conflicts in this area are that optimizations potentially can change program effect, there is no facility for machine code inserts, and the user has no control over open vs. closed subroutines.

k. Program Environment. The major conflict in this area is that JOVIAL does not have a special facility for including assertions in programs.

l. Translators. There are no major conflicts in this area.

m. Language Definition, Standards and Control. The major conflict in this area is the incomplete and ambiguous specification of the language.

3. Unnecessary Features in JOVIAL.

The following features of JOVIAL are unnecessary in the sense that they are not specifically needed to satisfy any of the Tinman requirements: the option of choosing between parallel and serial organization of tables, the SHIFT, SIZE, and DSIZE functions, unsigned integers, and the fall-through option of the SWITCH statement. We recommend that the

fall-through option of the SWITCH statement be deleted (due to its error-proneness) and the other features named above be retained.

4. Recommendations concerning JOVIAL.

On the basis of the evaluation conducted in this chapter, we conclude that it is inadvisable to modify JOVIAL to satisfy the Tinman. JOVIAL differs substantially from the Tinman with regard to the specific requirements and also the general goals and philosophy. Consequently, it is not reasonable to expect that JOVIAL could be modified to satisfy the Tinman while still adhering to its original design goals.

CHAPTER 6

FORTRAN EVALUATION

Section I. LANGUAGE SUMMARY

1. Lexical Properties.

a. FORTRAN is a semi-fixed format, line-oriented language. End-of-line flags the termination of a statement unless a non-zero digit appears in the continuation column, 6, of the succeeding line. Columns 1 through 5 are reserved for an optional numeric label, and the statement itself must appear somewhere within columns 7 to 72.

b. The language is defined using 49 characters, all of which appear in the 64 character ASCII subset. Non-standard symbols (i.e., any an implementation wishes to support) may appear within character data, edit field descriptors and comments.

c. Identifiers may be a maximum of 6 characters in length and must begin with a letter. Spaces are permitted internal to syntactic units as a break character and are ignored, except within keywords and character literals. Keywords are not reserved and their recognition relies on context.

d. Only full-line comments are supported. A "C" or "*" in column 1 signals a comment line.

2. Data Types

Figure 5 categorizes the FORTRAN data types.

a. Built-in Data Types.

(1) Introduction. FORTRAN differentiates between two classes of built-in data types according to their storage characteristics. Character variables occupy a fixed number of character storage locations determined by their compile-time specified length. All other scalar data types are allocated either one or two (for complex and double precision) non-character storage locations.

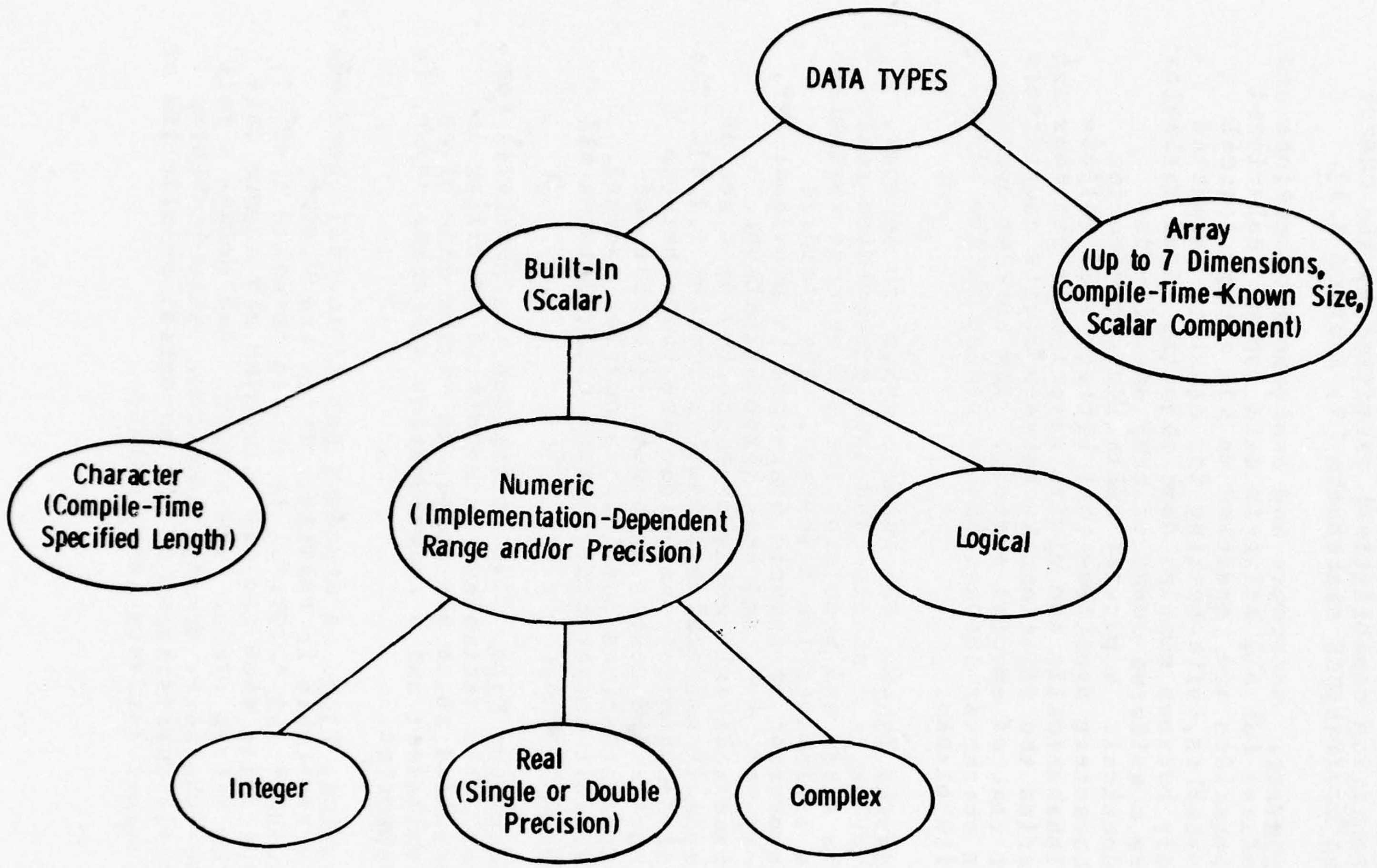

Figure 5. Data Types in FORTRAN

This distinction restricts the associations (i.e., overlaying capabilities) permitted by the COMMON and EQUIVALENCE statements [1, Section 8.3].

(2) Behavior. Reference and assignment operations are defined for all built-in data types. Relational operations are permitted on all except logical variables, with testing for equality permitted only between complex data objects. Real variables are considered equal if they are bit-wise identical. A partial collating sequence for characters specifies that letters will collate alphabetically and digits numerically but does not define the relationship between digits and letters or that of special symbols. The smaller operand in character comparisons is padded on the right with blanks.

(3) Numeric Types. The numeric types are integer, complex, and single- and double-precision reals. The range and precision for all numeric variables is implementation-dependent. The standard assortment of numeric operators is provided: "+", "-", "*", "/", and "**" (exponentiation). Mixed-mode arithmetic is supported by a set of implicit conversion rules [1, Section 6.1.4]. The only prohibited operations are those between complex and double precision variables and exponentiations involving a complex operand. Explicit conversion routines exist between all numeric types.

(4) Character Type. Two operations are permitted for character variables. A substring specifier is supplied which may appear on either side of an assignment and a concatenation operation, "//", is supported.

(5) Logical Type. A standard set of logical operators is available in FORTRAN. These are ".OR.", ".AND." and ".NOT.". It should be noted that [1] specifies that logical variables may assume only the values .TRUE. and .FALSE. and occupy a full non-character storage location. This prohibits their capitalizing on the potential parallelism of logical hardware instructions.

b. Arrays. The only aggregate data structure available in FORTRAN is the array. Arrays must be of compile-time size and dimensionality. The default lower bound on each dimension is 1, but may be overridden by the programmer. The array permits homogeneous composition of any of the built-in data types. Elements of a character array must all be of identical length. Actual array parameters must be equal to or greater than their associated dummy parameters in size and are always passed by reference. Component selection is the only array operation supported.

3. Procedures.

a. Procedure Classes. FORTRAN supports three types of procedures. Statement functions, similar to mathematical functions, are limited to the computation of a single statement and return a value. They are local to the program module (*) in which they appear, may access any variables defined within that program module, may not have side effects, and contain parameters local to their single statement. Function subprograms also return a value but may be of arbitrary length, and can access variables passed to them as parameters, available through COMMON, or defined locally to the subprogram. Subroutines are identical to function subprograms but may not return a value and are invoked by a CALL statement.

b. Parameter Types.

(1) Value parameter. Any built-in type may be passed "by value". The specification of value passing occurs on the calling side of a procedure invocation. For example, the parameters "+R" imply a "by value" or copy-in mechanism, whereas "R" denotes "by reference."

(2) Reference parameter. All data types including arrays may be passed "by reference."

(3) Procedure parameter. Either a function subprogram or subroutine may be passed as a parameter. An EXTERNAL declaration is required for all such

----

(*) A program module is either a main program, function suprogram, or subroutine. The most local scope possible for a variable is a program module. The only exception to this rule is parameters to statement functions.

parameters.

(4) Alternate return specifier. Statement labels may be parameters to subprograms. Optionally, a RETURN statement which is used to transfer control back to the calling program module may specify a positive integer which will cause control to return to the nth label parameter, where n is the value of the integer so specified.

c. Recursion. Recursion is not permitted in FORTRAN.

4. Statements

a. Assignment Statement. The FORTRAN assignment operator is the equals sign, "=". Type compatability is required for all assignments but implicit conversions will be automatically made between numeric types. Array assignment is not available.

b. CONTINUE Statement. The CONTINUE statement causes no action to be taken but can contain a statement label and thereby specify a control point.

c. GOTO Statement. The FORTRAN GOTO permits control transfers to any label local to the containing program module.

d. END Statement. This statement denotes the lexical end of a program module.

e. DO Statement. Iterative execution in FORTRAN is provided by the DO statement. A loop control variable, which is global to the containing program module, is required. Its initial value and termination limit must be specified. An optional step size may be included but will default to 1 if omitted.

f. Computed GOTO Statement. The effect of a case statement is provided in FORTRAN by the computed GOTO. It transfers control to one of a list of labels according to the value of an integer expression.

g. CALL Statement. The CALL statement is used to invoke a subroutine.

h. RETURN Statement. The RETURN statement was discussed in paragraph 3.b(4) of the section.

i. ENTRY Statement. The definition of alternate entry points to subprograms may be done with this statement. ENTRY statements may define dummy parameters in the same manner as subroutine or function specifiers. Appropriate rules exist to prohibit reference to an undefined dummy parameter within subprograms [1, Section 15.7].

j. SAVE Statement. This statement specifies variables within a subprogram which are to retain their values from one invocation to the next. These variables then become identical to "own" variables in ALGOL-60.

k. ASSIGN Statement. The ASSIGN Statement allows the assignment of a statement label to an integer variable.

l. Assigned GOTO Statement. Once a label has been assigned to an integer variable, this statement permits control transfer to that label. Optionally, a list of potential destinations may be included.

m. Conditional Statements.

(1) Arithmetic IF. Depending upon the relationship between a specified arithmetic expression and zero, this statement causes a jump to one of three labelled locations.

(2) Logical IF. This statement allows the value of a logical expression to cause conditional execution of a given statement.

n. EQUIVALENCE Statement. Variables of similar storage type (i.e., character or non-character), including arrays, may be EQUIVALENCEd. Variables so associated share common storage locations.

o. COMMON Statement. The COMMON statement allows inter-routine communication. COMMON blocks simply specify regions of memory upon which each routine is permitted to define any organization. The only restriction that exists is that character and non-character variables may not appear within the same COMMON block or be EQUIVALENCEd (overlayed) by two organizations of a given block

5. Storage Allocation.

FORTRAN is designed so as to allow static storage allocation. The size of all data items, except arrays passed by reference (which do not require memory allocation), is known at compile-time. No recursion or parallel processing is permitted.

6. Files and I/O.

a. File Storage.

(1) Internal files. Each internal file is associated with some area of character storage. This allows I/O to be directed to or from a character datum.

(2) External files. External files reside on an external device identified by an integer unit number. The association of a specific device with a unit number is implementation-dependent.

b. File Types.

(1) Sequential files. Sequential files contain an ordered sequence of records. Each record is stored in some formatted fashion. Records may be of differing formats and are read or written sequentially.

(2) Direct files. Direct (random) access files consist of a set of records of equal size. Associated with each record is a record number which permits retrieval and update to occur in any order.

(3) Stream files. Stream files are unformatted, read sequentially, and treated as a stream of characters.

c. I/O Control.

(1) External files are selected for access by the OPEN statement. Each file can, optionally, have associated with it an implementation-dependent file name. The OPEN statement permits specification of accessing method, record length,

the maximum number of records, treatment of blanks, and file status. The file status may be one of the following: OLD, NEW, SCRATCH, or UNKNOWN. The UNKNOWN value has an implementation-dependent meaning.

(2) Data transfer is accomplished by the READ, WRITE, and PRINT statements. These statements may include a unit number and format reference. The absence of a unit number has an implementation-dependent meaning. If the format specifier is omitted, list-directed formatting is implied. List-directed formatting is available only for stream files. (See [1, Section 13.6] for the details of this formatting.)

(3) Three statements are provided for file positioning: BACKSPACE, REWIND, and ENDFILE. Only the ENDFILE statement is available for use with direct files.

7. Process Scheduling.

FORTRAN contains no facilities for process scheduling.

8. Exception Handling.

The exception handling facilities available in FORTRAN are limited to error conditions identified during input and output. FORTRAN I/O statements permit the optional inclusion of error labels. The situations in which this error path will be taken are implementation-dependent. No facility is defined in [1] for providing information to the program about the type of error encountered. The action taken in the event of any non-I/O error is entirely implementation-dependent.

9. Compile-Time Facilities.

[1] does not define any compile-time facilities. The language contains no macros, compile-time variables, or conditional-compilation capabilities. A separate compilation facility is implicit in the EXTERNAL statement but is left undefined.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. The types of all FORTRAN data objects are determinable at compile-time and are unalterable at run-time. However, explicit type declarations are not required. The implicit specification is that symbols beginning with the letters I, J, K, L, M, N represent integer variables and all others denote floating-point variables. The programmer may both redefine this implicit assumption and override it through explicit declarations.

c. The notion of "type" in FORTRAN is somewhat limited. For example, the type of an array is taken to be the type of the underlying component (integer, real, etc.) and does not include the dimension sizes or even the number of dimensions. Thus FORTRAN does not prohibit the programmer from passing an array argument to a routine that expects a differently structured array parameter. As stated in [1, Section 15.9.3.3], "the number and size of dimensions in an actual argument array declarator may be different from the number and size of the dimensions in an associated dummy argument array declarator."

d. In general, FORTRAN does not require the translator to validate the type compatibility of associated (*) data items. "A processor assumes that the type specified for the name agrees with the type of the data" [1, Section 20.4]. This statement demonstrates the weakness of FORTRAN's type-checking. The responsibility for adherence to type rules rests with the programmer.

e. The implicit assumptions made by FORTRAN may be easily removed. The mechanisms for explicit type declarations already exist. The only change required involves reporting undeclared variables. This simply implies modification of the symbol table maintenance routines.

(*) Two data items are said to be associated if any component of one shares a common storage location with some component of the other. The mechanisms which cause association in FORTRAN are parameter passing, COMMON and EQUIVALENCE statements.

f. The strengthening of the concept of array type in FORTRAN may be accomplished fairly easily. It will, however, require some runtime checking to be done. This is necessary when array dimensions are specified dynamically (i.e., in terms of some parameter) in a subprogram. This will eliminate some common programming tricks of the language, but should not severely limit its programming power, especially if array operations are permitted (see our discussion in connection with B1).

g. The validation of type compatibility for all associated data items requires extensive modification of the language.

(1) All formal and actual parameters must be checked for compatibility. To support full compile-time checking would necessitate further specification of procedure parameters. The EXTERNAL and INTRINSIC declarators need to be expanded to include definition of the type and number of parameters for the associated procedures.

(2) The COMMON and EQUIVALENCE statements would be the source of the greatest problems in regard to strong type-checking in FORTRAN. One has basically three options as to how to overcome these problems. These statements can be totally eliminated from the language; full type-checking can be performed on them; or some compromise can be made regarding their type security. If removed from the language these statements could be replaced by secure access and overlaying facilities. Strong type-checking would ensure the security of these statements but limit their use for overlaying purposes. One possible compromise would be to limit the access scope of differently-typed associated items and enforce the restrictions on the accessing of these variables before assignment of the proper type.

2. Data Types (A2).

a. Degree of compliance: P

b. FORTRAN provides data types for integer, floating point, Boolean and character [1, Chapter 4] and provides

arrays of these types [1, Chapter 5]. There exists no general fixed-point type besides integer, and there exists no record definition facility, a serious drawback.

c. The feature of FORTRAN which most closely approximates a record definition facility is the named COMMON statement [1, Section 8.3]. Several modifications could be made to transform this statement into a rudimentary record declarator; however, these would significantly change the intent of this feature.

d. The introduction of a record definition facility in FORTRAN would require extensive additions to the language. While these changes would not necessarily prompt modification of existing features, they would certainly complicate their implementation.

e. A record consists of the composition of a finite number of components, each of some predefined type and size. The record itself and each component has an associated name by which it can be referenced. For consistency with E1, the declaration of a record should not require any storage allocation, but should simply serve as a template for variables which are later defined to be of that type (i.e., the declaration of a record defines a new type).

f. The definition of a record facility will interact with the existing COMMON and EQUIVALENCE statements. If components within a record can be EQUIVALENCEd, type-checking will be complicated but alternate structures could be so supported (see A7 for further discussion). At present, the use of character variables in common blocks is restricted. "If a character variable or character array is in a common block, all of the entities in that common block must be of type character" [1, Section 8.3.1]. This restriction stems from a need to prevent implementation-dependent common block associations between numeric and character variables. Strong type-checking on COMMON variables (or their elimination) would circumvent this problem. Otherwise, an implementation-independent character representation and some functional relationship between character and non-character storage sizes could be defined.

g. The introduction of a general fixed point facility is discussed in A4.

3. Precision (A3).

a. Degree of compliance: F

b. FORTRAN offers "REAL" and "DOUBLE PRECISION" as the only floating-point precision specifications. Both have a hardware-dependent definition [1, Section 4.5]. The precision may be defined for a program unit by using the IMPLICIT definition to specify all variables of floating-point type to have single or double precision. This implicit specification may be overridden in explicit type specifications.

c. To bring FORTRAN into partial compliance with this requirement would be relatively easy. It could be accomplished simply by requiring a statement of the intended meaning of single and double precisiion in each program. This would promote portability and necessitate only minor modification of the language (the inclusion of one new statement) and the translator (at least, informing the user if the machine cannot support that precision).

d. The inclusion of an ability to further delineate precision specification for all variables would be slightly more difficult but still quite feasible. The rules for evaluation of mixed precision expressions already exist [1, Section 6.1.4]. Either existing type declarators would have to be extended or a new precision specifier added to permit further precision definition. The translator could simply use this precision definition to decide whether a variable should be represented in conventional single or double precision. This would involve an extremely minor change.

4. Fixed Point Numbers (A4).

a. Degree of compliance: F

b. FORTRAN does not offer any fixed-point data type except integers. Integers are treated as exact quantities with a step size equal to one.

c. The introduction of a general fixed-point data type would require a large effort. The language would need to be expanded to include a fixed-point declarator. If fully integrated into the present language, fixed-point variables, just as those of any other numeric type, would be permitted

to appear anywhere in arithmetic expressions. This would require more general mixed-type expression evaluation rules, additional intrinsic conversion routines, and compiler and run-time scaling procedures. Run-time scaling would be required when assigning floating-point values to fixed-point variables. In general, mixed-mode arithmetic would be greatly complicated, and possibly large efficiency losses, which are hidden from the programmer, could result. Another option is the elimination of mixed-mode arithmetic (see our comments concerning B8).

5. Character Data (A5).

a. Degree of compliance: F

b. FORTRAN makes the following restrictions on the collating sequence [1, Section 3.1.5]. "A" is less than "Z" and letters collate in alphabetic order; "Ø" is less than "9" and digits collate in numerical order; blank is less than "Ø" and less than "A". There are no other constraints on the collating sequence. This means that numeric characters may collate before (USASCII) or after (EBCDIC) alphabetic characters, and the order and position of special characters relative to alpha or numeric characters is undefined. There is a recommendation offered: "A processor, if possible, should use the ASCII collating sequence for the complete FORTRAN character set" [1, Section 2Ø.3]. FORTRAN offers neither standard translate routines nor alternate character sets.

c. The new draft proposal for FORTRAN improves FORTRAN's handling of character data, but the language still falls far short of meeting this requirement. To bring FORTRAN into total compliance would be a difficult task. FORTRAN does not offer any ability to define enumeration types. Even with the addition of such a facility (see B3), many problems still remain. For example, character variables and arrays are permitted to be used as format specifiers [1, Section 13.1.2]. The introduction of user-defined character sets would then greatly complicate input-output and cause the utilization of character variables as format specifiers to incur additional runtime costs.

d. Partial compliance can be gained, however, through a simple addition to the language. A statement should be added which permits the definition of any underlying

ordering assumptions that are being made in addition to those of the defining document [1]. Requiring use of this statement would improve both portability and readability.

6. Arrays (A6).

a. Degree of compliance: P

b. FORTRAN requires user specification of the number of dimensions [1, Section 5.1.5] and the range of subscript values of each dimension [1,Section 5.1.1.1]. The type of each array component may be specified implicitly [1, Section 5.2.1]. The number of dimensions and the array type are determinable at compile time. Subscript bounds must be known at compile time for arrays defined in the main program; the bounds for dummy arrays (i.e., formal parameters) may be specified at run time. FORTRAN arrays are allocated at compile time (or before execution of the main program). The default lower bound of an array is 1 [1, Section 5.1.1.2]. The range of subscript values for any given dimension is a contiguous subsequence of integers [1, Section 5.2].

c. Permitting the allocation of arrays at scope entry would entail major language modifications. The basic flavor of FORTRAN is one of static memory allocation. No features presently exist that require dynamic memory management. The addition of any such features would reduce runtime efficiency and greatly complicate the use of some existing facilities. The type-checking required by A1 would have to be done at run time. The correspondence of common block size would also have to be checked during execution. EQUIVALENCE statements could no longer be validated during compilation. These problems with COMMON and EQUIVALENCE statement stem from their storage orientation.

7. Records (A7).

a. Degree of compliance: F

b. FORTRAN has no facility for defining records (i.e., heterogeneous data structures with named fields); thus there is no provision for records with alternative structure. The way in which the facility called for in this requirement is achieved in FORTRAN is in fact the EQUIVALENCE statement prohibited by A7. The problem with EQUIVALENCE is that it

facilitates the defeat of type checking, as mentioned earlier in connection with A1. In addition, the feature has various restrictions [1, Sections 8.7.5 and 8.3.6] because of its storage orientation and the interactions with other features such as COMMON.

c. The introduction of an extremely primitive record facility was discussed in A2. The use of this facility in conjunction with a strongly-typed EQUIVALENCE statement permits safe use of alternate structures within records. The necessity of type compatibility could be removed if compile-time type-checking rules were relaxed. Performing the type-checking at run time would require the maintenance of a tag-field. The easiest way in which alternate structures could be supported would require all such structures to have the same size. This size restriction simply necessitates the allocation for the largest possible record. This would permit records to be stored in common without any further changes. Another alternative is dynamic allocation and common validation, which was shown to be a fundamental change in the language in A6.

Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. FORTRAN defines assignment and reference operations [1, Sections 10 and 2.12] for all scalar data types. However, assignment is not provided for arrays. Moreover, the absence of encapsulated type definitions implies that no user-defined assignment is permitted. It should also be pointed out that the label assignment allowed in FORTRAN (the ASSIGN statement [1, Section 10.31] is generally considered poor programming practice, since it serves to obscure the behavior of the program.

c. Array assignment poses no real difficulty for FORTRAN. Arrays are of compile-time-known size and dimensionality, except for adjustable arrays [1, Section 5.5.1] within subprograms which will be required to be of known dimensionality if the suggestions of A1 are implemented. This then leaves only the addition of runtime tests to check size compatibility when dealing with adjustable arrays and of code generation routines for array assignment.

d. Record assignment involves problems similar to those of array assignment. The required additional effort for its implementation depends upon how records are defined (see our discussions regarding A2 and A7). The ability to have run-time-determined alternate structures of different sizes would complicate assignment and reference operations.

e. The introduction of user-defined assignment for encapsulated type definitions is discussed in E5.

2. Equivalence (B2).

a. Degree of compliance: P

b. The FORTRAN comparison operator will compare any two scalar operands of arithmetic type or any two scalar operands of character type. If arithmetic operands are of different types then integer is converted to real or double precision, or real is converted to double precision. Two floating-point values are equal only if they are bit-by-bit

the same. FORTRAN does not provide a record data type and arrays may only be compared element by element.

c. What is required to bring FORTRAN into full compliance with this requirement is to extend comparisons to include array and record operands and to modify the equality rules for floating point numbers. In the case of arrays and floating-point numbers, meeting these requirements would entail only a small effort. Record comparison is somewhat more complex, and if alternate structures are to be supported, these comparisons would involve greater run-time and compile-time overhead. The inclusion of precision specifications as proposed in A3 would be the only design modification required to support limited-precision identity for floating-point numbers.

3. Relationals (B3).

a. Degree of compliance: P

b. All six relational operators are defined for numeric data and for character data [1, Section 6.3]. FORTRAN provides no facilities for definition by enumeration.

c. The inclusion of facilities for definition by enumeration requires extension of FORTRAN, but poses no great difficulties if strong type-checking is maintained. Elements of enumerated types can be represented as integer quantities denoting their position in the enumeration. This permits all six relational operators to be applied in exactly the same manner as they would be for integers.

4. Arithmetic Operations (B4).

a. Degree of compliance: PT

b. Aside from lacking integer division with a real result, FORTRAN provides the required operations [1, Section 6.1]. The remainder of integer division is calculated with the MOD intrinsic function [1, Section 15.10]. Floating point operations are as precise as the operand of greater precision [1, Section 6.1.4].

c. Integer division with a real result could very easily be explicitly defined in the language, but it is already indirectly available. By first converting the operands to real numbers, one is able to compute the required division.

d. The present division operator could be redefined to return a real result with integer operands; however, this could lead to great confusion. Historically, FORTRAN has truncated the result of such an operation, and many programmers are accustomed to using this feature.

5. Truncation and Rounding (B5).

a. Degree of compliance: PT

b. FORTRAN satisfies this requirement, except for the lack of information on the behavior of the program in the presence of arithmetic overflow. It is possible for such an overflow to result in high-order truncation with no message to the user that this has occurred.

c. Overflow reporting is dealt with as part of the general discussion of exception handling in G7.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. The FORTRAN built-in Boolean operations include "and", "or", and "not", but "nor" is not provided [1, Section 6.4]. FORTRAN allows but does not require short-circuit mode in the evaluation of "and" and "or" [1, Section 6.6.1].

c. The addition of a "nor" operator to the language is trivial. It simply requires an addition to the table of relational operators in [1, Section 6.4.1] and a few new lines of code to the compiler. The strengthening of wording in [1, Section 6.6.1] to require short-circuit mode evaluation will not only bring FORTRAN into full compliance but will also simplify the language. As it stands, not only can the programmer not depend on short-circuit evaluation, but he must also accept the fact that variables assigned to as a side effect of a possibly short-circuited expression become undefined after evaluation of that expression [1, Section 17.3].

7. Scalar Operations (B7).

a. Degree of compliance: F

b. FORTRAN does not permit any operations on data aggregates except I/O operations [1, Sections 6.1.2.1 and 10.1]. All operations required in B7 must be explicitly programmed element by element. FORTRAN does not provide records.

c. Record definition is discussed in A2 and A7. Assignment of data aggregates is reviewed in B1.

8. Type Conversion (B8).

a. Degree of compliance: P

b. FORTRAN converts implicitly among arithmetic types in expressions and for assignments [1, Section 6.1.4]. FORTRAN does not convert an argument to the parameter type; when arguments and parameters differ in type the program is in error and this error need not be detected [1, Section 15.5.2.2 and 15.6.2.3]. FORTRAN does provide all the required explicit conversion operations using intrinsic functions among arithmetics [1, Section 15.10] and I/O operations between object and character representations or arithmetic data.

c. The type-checking proposed in A1 would guarantee the reporting of mismatched actual and formal parameters. FORTRAN could be modified fairly straightforwardly so as to remove all implicit conversions. The routines for all valid explicit conversions exist. However, this would be a fairly radical departure from the flavor of the language.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: P

b. FORTRAN does not provide any facilities for restricting the numeric ranges offered by a hardware implementation; therefore, conversion will never be required. There is no provision for detecting integer overflow or floating point overflow or underflow.

c. Range specifications can be added to FORTRAN. No features of the language directly prohibit this, but if the compiler is meant to use these specifications to optimize memory utilization, several features will be impacted. If the storage size for variables is to be compiler-determined,

then statements which are storage oriented (i.e., EQUIVALENCE and COMMON) must be handled delicately. Integer and fixed-point truncation tests could be optionally compiled into the object code and when truncation occurs processed like any other exception condition (see G7).

10. I/O Operations (B10).

a. Degree of compliance: P

b. FORTRAN provides operations allowing programs to interact with files, channels and devices including terminals [1, Chapter 12]. Data (and control information in the form of data) may be sent and received. FORTRAN also provides some explicit I/O control functions [1, Section 12.10]. The language allows user control over some exception conditions; however, the definition of an exception condition is implementation dependent, and whether or not the user retains control of all exception conditions is an implementation decision. The form of file names and the possible characteristics of files are implementation dependent.

c. To fully comply with requirement B10, greater standardization is required in FORTRAN I/0. A full definition of the OS interface is needed. This would necessitate further work on the language definition and an additional large effort in compiler writing, especially if the possibility of running with no operating system is allowed for.

d. A small modification specifying that all errors would transfer control to the error label in an I/O statement and requiring that such a label exist would bring FORTRAN into greater compliance with this requirement. Additionally, some mechanism for returning the nature of some otherwise undefined error should exist. Both these features could be incorporated into the language with small modifications to the defining document and a reasonable amount of added effort in compiler implementation.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. FORTRAN provides no facility for defining enumeration types; however, the FORTRAN logical array can be used to provide set operations.

c. The primary difficulty with using logical arrays to simulate power sets is that an element of such an array occupies a full noncharacter storage cell [1, Section 4.7]. This precludes the use of logical machine instructions for parallel processing of these operations. Either of two options can be chosen to overcome this problem. A logical array can be defined to be optionally packed or a set declarator can be added to language. The use of a packed logical array, while requiring less language modification, would probably lead to implementation-dependencies. This would be due to the dictating of array size by machine word size (e.g., a 36 element set would occupy 2 words on an 18-bit machine and only 1 on a 36-bit machine).

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

   a. Degree of compliance: F

   b. FORTRAN explicitly prohibits certain side effects [1, Section 6.6]:

> "The execution of a function reference in a statement may not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement may not alter the value of any entity in common ... that affects the value of any other function reference in that statement."

Other side effects, such as overflow, are ignored. Moreover, there is no guarantee that occurrences of the prohibited side effects will be detected in programs. There is no guarantee that side effects will be evaluated in left-to-right order, especially since the order of evaluation in expressions is left undefined.

   c. Satisfaction of this requirement within FORTRAN would affect only the optimization section of the compiler. However, in this area it would have tremendous impact. Because of the difficulty of detecting all side effects, many optimizations commonly applied to FORTRAN programs would be eliminated. This could have significant impact on run-time efficiency.

   d. If this requirement is truly the desired goal then the restrictions can be easily included in the language definition. Compiler implementation need be no more complex, since potential optimizations can be ignored if their application might violate C1.

2. Operand Structure (C2).

   a. Degree of compliance: PT

   b. FORTRAN has few levels of operator hierarchy, and the ones present are widely recognized. However, FORTRAN does not require explicit parentheses "when the execution order

is of significance to the result within the same precedence level". For example, X/Y/Z, X/Y*Z, and X**Y**Z are legitimate expressions, with interpretations (X/Y)/Z, (X/Y)*Z, and X**(Y**Z), respectively. The FORTRAN user is not able to define new operator precedence levels nor to change the precedence of existing operators.

c. The only required modification here is minor. The rules for evaluating unparenthesized expressions containing operators of equal precedence should be dropped and the language definition should state, "when the order of evaluation is not determinable from operator precedence and can be significant to the result, the expression must be more fully parenthesized". Checking for this condition would constitute a fairly trivial modification to the compiler.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. FORTRAN completely satisfies this requirement, correcting the deficiencies (cited in the Tinman) which were characteristic of previous versions of the language.

4. Constant Expressions (C4).

a. Degree of compliance: P

b. FORTRAN allows constant expressions in programs anywhere that constants are allowed. The FORTRAN standard does not require constant expressions to be evaluated at compile time.

c. The inclusion of a rule that constant expressions must be evaluated at compile-time entails little additional effort. The impact on compiler design is relatively small. Even in cases of cross-compilation, as compared to the overall compiler development, this necessitates a reasonably small amount of work. The ease with which this requirement can be met stems from its independence from other language features.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: F

b. FORTRAN does not have exception handling or a parallel processing capability. The intrinsic functions of FORTRAN obey the same parameter-passing rules as user-defined functions. There are no operations applicable only to parameters.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. FORTRAN requires that the type of each argument agree with the type of the corresponding parameter. The bounds of an array may be passed explicitly as arguments to a procedure. Whether a type transfer hidden in a procedure call is or is not detected, and the result of any operations involving such type transfer, are implementation dependent. In addition, FORTRAN's restrictions on array passing are fairly weak [1, Section 5.5]: "If the actual argument is an array name, the size of the dummy array must not exceed the size of the actual argument array."

c. The language and compiler modifications required for strengthening parameter type checking are discussed above in connection with requirement A1.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. FORTRAN partially satisfies this requirement. Scalars may be passed either by value or by reference [1, Section 15.9.3]; however, the kind of passage is specified on the calling side and not with the formal parameter. For example, A is passed by reference in CALL SUB(A) and by value in CALL SUB ((A)) or (if A is numeric) in CALL SUB(+A) [1, Sections 20.6 and 15.9.3]. Arrays may only be passed by reference. FORTRAN provides procedure pamameters and also "alternate return specifiers" which may be used to control action when user-detected exceptions occur.

c. The ability to pass composite structures (i.e., records and arrays) by value would require a good amount of

run-time support. Since arrays may not be of compile-time determined size (in the case of adjustable arrays), some dynamic memory management would be required. This leads to the problems discussed above in connection with requirement A6.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: F

b. FORTRAN fails to satisfy this requirement. Any procedure parameters which are not explicitly declared are specified by default and do not provide the required facility.

c. It would be possible to support generic procedures in FORTRAN; however, several major problems would arise from their inclusion. Some mechanism must be established for their definition. This mechanism, while potentially syntactically quite similar to normal subprograms, would be semantically quite different. Semantically, it is most like a macro definition (but with only limited replacement and with only implicit as opposed to explicit information transfer).

d. This feature could be included in the language with a moderate amount of language redesign and a large degree of compiler support.

9. Variable Number of Parameters (C9).

a. Degree of compliance: F

b. As stated in [1, Section 15.9], "The number of actual arguments must be the same as the number of dummy arguments in the procedure referenced." FORTRAN has no provision for variable numbers of parameters, except for the built-in ("intrinsic") functions MAX and MIN.

c. To provide the suggested facility would require a moderate amount of work. The method by which arrays are passed in FORTRAN somewhat simplifies the task. Since "the size of an adjustable array must not exceed the size of the actual argument array," the mechanism for accepting a variable number of elements without setting any maximum already exists. However, the consequence of using this

array passing mechanism is that all arguments so specified appear as by-reference parameters at procedure entry. On the calling side, however, since arrays must be stored contiguously in memory and the parameter variables as declared may occupy non-contiguous locations, a block of storage must be allocated and the values copied into it. This implies, despite the apparent by-reference passing on the procedure side, that by value parameter passing needs to be used. The aforementioned allocation does not necessitate dynamic memory management since the actual number of parameters is known at compile time.

Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: PT

b. FORTRAN satisfies this requirement to a high degree [1, Section 8.6]; the only exception is that there is no provision for constant arrays.

c. The inclusion of an array literal as discussed in D2 and the ability to use it within the PARAMETER statement [1, Section 8.6] will bring FORTRAN into total compliance with this requirement. This constitutes a relatively minor modification and incurs no run-time costs.

2. Numeric Literals (D2).

a. Degree of compliance: P

b. FORTRAN provides a syntax and consistent interpretation for constants of built-in data types [1, Sections 4.3 through 4.8]. The relationship between values of numeric literals and input or output data is not considered in [1].

c. If array assignments as reviewed in B1 are permitted, only a small expansion of the language to include array literals is needed. The requirement for equivalence between numeric literals and input data must be added to [1]. This introduces the difficulties mentioned in the Tinman when cross-compilation is involved but the additional work relative to the overall development of a cross-compiler is small.

3. Initial Values of Variables (D3).

a. Degree of compliance: P

b. FORTRAN permits the user (via the DATA statement [1, Chapter 9]) to specify the initial values of individual variables. The initialization specification is separate from the declaration and, for COMMON variables, may appear in a different program unit [1, Chapter 16]. All FORTRAN variables are preallocated; thus, variables are initialized at the time of their apparent allocation. There are no

default initial values [1, Section 2.11]. FORTRAN has no provision for testing for variable initialization.

c. The ability to initialize variables as part of their declaration requires a syntactic modification of FORTRAN; however, it should only shift where the work is done in the compilation not increase it. This necessitates moderate language redesign and only small compiler changes.

d. Detection of potentially undefined references at compile time and run-time testing for initialization would increase the complexity of the compiler but would not change the nature of the language. [1] states, "an entity must be defined at the time a reference to it is executed." The additions necessary to enforce this restriction would require greater work during compiler development and be somewhat complicated by the COMMON and EQUIVALENCE statements. This would be particularly true if these statements are permitted to associate variables of differing types.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. FORTRAN does not require its users to specify the range of numeric variables nor does it provide any mechanism for such specifications. The only range limitations in FORTRAN are those inherent in the hardware; the results of any operations exceeding hardware limitations are undefined [1, Section 1.2.2].

c. The scope of modifications needed to support a fixed-point number facility is discussed above in connection with requirement A4. The introduction of a mandatory range specification for all numeric variables can be accompished without adversely impacting other language features. However, its use for optimization of storage allocation would cause intricate problems in regard to the COMMON and EQUIVALENCE statements. These problems could be minimized by a redesign of these statements in such a manner as to make them less storage oriented. This would entail a large amount of effort in both language redesign and compiler development.

5. Variable Types (D5).

a. Degree of compliance: P

b. FORTRAN provides neither type-definition facilities, record data structures, nor enumeration types. Arrays and variables may be of any built-in data type; the number of array dimensions is limited to 7 [1, Section 5.1.1].

c. Full compliance with D5 requires extensive language modifications. Some of the issues involved have already been discussed (records in connection with A2 and A7, enumeration types in connection with B3). The ability to define arrays of any component type would complicate the subscripting rules as described in [1, Section 5.4]. Removing the restrictions on the dimensionality of arrays would be a relatively minor modification.

6. Pointer Variables (D6).

a. Degree of compliance: F

b. FORTRAN has no true pointer facility. The EQUIVALENCE statement provides some of the attributes of such a facility (e.g., shared substructure, acting as an additional reference to a datum) but due to its storage orientation is quite different in nature. Many of the issues regarding pointer variables concern features not present in FORTRAN (e.g., composite types). The ease with which pointers can be introduced depends on how these features are designed. A clean implementation should facilitate the introduction of pointers; however, this is a major modification to both language and implementation.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of complaince: F

b. FORTRAN is quite weak in the area of definition facilities. The language permits functions and subroutines to be defined by the user, but there is no provision for defining new infix operators. The data structuring mechanisms are limited; there are no capabilities for user-defined data types, and the only data structuring facilities available are the array and the EQUIVALENCE statement.

c. Many features needed for data definition have already been discussed (records in connection with A2 and A7, enumeration types in connection with B3, and composite component arrays in connection with D5). As can be seen, these modifications alone constitute extensive changes. To reach full compliance with E1 would require a total language redesign. A data definition facility interacts with almost all other language features (e.g., parameter passing, reference and assignment operations) and requires constant attention throughout language definition.

d. The effect of such extensive modifications on implementation would be twofold. The compiler necessary to support these new features would be several times as complex as present compilers, increasing both the cost and time of development. The size of the resulting compiler would severely limit the number of machines on which it could be supported.

e. The specific aspects of data definition introduced in Tinman requirement E1 will be dealt with in the other paragraphs of this section. However, let us mention here that their dependence on the aforementioned language design is great.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. Since FORTRAN contains no type definition facilities as discussed in E1, it fails to comply with this

requirement. Therefore, in terms of modification this reduces to an added condition on the redesign necessary to satisfy E1. As such the additional effort relative to the major modifications already needed is small. Within the scope of the present language, facilities for performing operations on arrays are required. These have been discussed in B1 and B2.

3. No Default Declarations (E3).

a. Degree of compliance: PT

b. The FORTRAN default (symbols beginning with the letters I, J, K, L, M, or N stand for integer variables or functions, and all other symbols stand for single precision real variables or functions) is well documented in the language standard and widely known. It is, however, a default. Symbols whose type differs from these defaults must be explicitly declared. All other program components must be explicitly defined.

c. The modifications necssary for removal of FORTRAN's implicit type defaults are described in A1.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. FORTRAN offers no such facilities.

c. This requirement constitutes an added goal of the language redesign required by E1. Operator extension is, however, somewhat more limited in scope than the general changes described above. If mixed mode expression evaluation is eliminated from the language (as required by B8) and strong type checking is performed at compile time (as required by A1), the scope of additional modifications is fairly small and no large problems should be encountered during implementation.

5. Type Definitions (E5).

a. Degree of compliance: F

b. FORTRAN offers no facilities for defining new types and thereby totally fails to meet this requirement. The

scope of modifications needed to support such facilities is prohibitive (see our discussion above concerning E1) and would be further complicated if compliance with E5 is desired.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: F

b. FORTRAN does not offer any mechanism for defining new types by enumeration, by discriminated union, or as power sets of enumeration types. The Cartesian product of existing types is provided in the array definition facility (to a maximum of seven dimensions). The effect of power sets can be partially realized in FORTRAN via logical arrays.

c. The introduction of general type definition facilities has been considered in connection with E1. Several of the mechanisms suggested in E6 have been discussed in regard to FORTRAN in other sections of this report (records in connection with A2 and A7, enumeration types in connection with B3 and use of logical arrays to simulate power sets in connection with B11). Discriminated union could be supported but would greatly complicate type checking and necessitate some run-time type validation.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The EQUIVALENCE statement in FORTRAN has the effect of a free union feature. FORTRAN has no subsetting provisions.

c. The problems associated with the EQUIVALENCE statement have been discussed in connection with A7.

8. Type Initialization (E8).

a. Degree of Compliance: F

b. Lacking type definition, FORTRAN provides no facilities for meeting this requirement.

c. This requirement is actually an extension of the facilities required by E5. The problems associated with its realization are similar to those already discussed above in connection with requirement E5.

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

a. Degree of compliance: P

b. FORTRAN offers most of the namescope control required by F1. Variables may be defined local to some subprogram and either be saved from one activation to another or independently allocated for each call. However, FORTRAN is not a block-structured language; thus, a name is accessible throughout the entire subprogram in which it appears. Statement-function parameter names are an exception to this rule; they are accessible only in the single statement-function statement [1, Section 2.9].

c. FORTRAN provides the COMMON statement to allow the definition of inter-routine access rights. Named common statements permit the programmer to limit access to a specific set of routines. This mechanism has at least one great weakness: the common name identifies some region of memory but not the organization of that region. The associated type-checking problems have been discussed in connection with A1.

2. Limiting Access Scope (F2).

a. Degree of compliance: F

b. In FORTRAN, variables are either local or potentially global. No mechanism is available for limiting accessing privileges. Once a variable is defined in COMMON, any and all routines can obtain access to it. The optional inclusion of COMMON declarations allows limiting of access on the call side. The mechanism by which COMMON statements are declared also permits the local renaming of variables. However, the thrust of this requirement is in the direction of user-defined types, and due to FORTRAN's lack of type definition facilities it fails to comply. Inclusion of the ability to redefine substructure names and specify limited global accessing rights with regard to the COMMON statement would entail a reasonable effort. The implementation of such facilities would be non-trivial.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: P

b. Identifiers need not be declared if the user accepts the FORTRAN defaults. Multiple use of variable names is not allowed in the same scope. Variable names may be the same as named-COMMON names [1, Section 20.8]. FORTRAN keywords are not reserved and may, in some contexts, be taken for variable names. FORTRAN is not a block-structured language. Subprograms are always an access scope and the most local definition applies to identifiers. The scope of identifiers is wholly determined at compile time.

c. The reservation of keywords and forced uniqueness of common and variables names constitute a minor change in the language definition. In fact, if keywords become reserved, compiler implementation would be simplified.

d. The changes necessary to make FORTRAN block structured ("lexically embedded scope rules") would be fairly large and involve extensive additions to the compiler. Impact on other language features would be primarily restricted to the common statement. Some consistent and convenient rules would have to be developed to define the interactions between this statement and the block structure.

4. Libraries Available (F4).

a. Degree of compliance: T

b. FORTRAN offers a large set of mathematical functions, and specialized application-oriented subroutine libraries may be made available for FORTRAN. These libraries are outside the scope of the FORTRAN standard definition, but there is nothing in the language which prevents their development.

5. Library Contents (F5).

a. Degree of compliance: F

b. The FORTRAN definition [1] does not directly specify a library facility. The EXTERNAL and INTRINSIC statements do, however, imply that some sort of separate compilation facility must exist.

c. The inclusion within [1] of a definition of a library feature would entail only a small amount of work. The necessary additions to provide compiler support are fairly minor due to the simplicity of FORTRAN's parameter passing and subroutine interfaces. The additions would have no significant impact on existing language facilities.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: F

b. FORTRAN has no compool facility, nor are compile-time libraries provided.

c. The addition of compools is dependent upon the manner in which data definition facilities are made part of the language (see our discussion above concerning E1). The addition of libraries and compools should be fairly simple in the context of present language facilities (see our discussion above concerning F5), but interaction with other needed additions for Tinman conformance could cause complex problems.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. The FORTRAN user has two general I/O statement forms: READ and WRITE. Additional I/O statements such as OPEN, CLOSE, BACKSPACE, REWIND, and ENDFILE are valid only for some kinds of files and devices. File processing may be either stream (character-oriented) or record (record- or block-oriented). Any device which has requirements not satisfiable with these operations requires special-purpose subroutines written for its service.

c. These issues are addressed in our discussion of requirement B10.

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. FORTRAN provides mechanisms for sequential, conditional, and iterative control. Recursive control is not allowed [1, Section 15.5.3]. No control structures exist in FORTRAN for parallel processing (pseudo or otherwise), asynchronous interrupt handling or arithmetic exception condition (i.e., overflow, divide by zero, etc.) processing. For certain types of I/O exception conditions, control is available to the user.

c. FORTRAN control mechanisms are separable and the user need not pay for unused capabilities. However, FORTRAN control mechanisms are not "composable" in the sense of G1 and, in fact, the DO statement of the previous FORTRAN standard cannot now be composed in FORTRAN [1, Section 11.6.3] because the condition is now tested before the first iteration.

d. The scope of modifications required to bring FORTRAN into compliance with the various aspects of G1 is discussed point by point in the succeeding paragraphs of this section. The scope of the changes taken as a unit is quite large. Implementation would be greatly complicated. Ad hoc addition of (pseudo) parallel processing, recursion, and exception and interrupt handling does not seem feasible. The facilities must be considered throughout the language design process and impact almost all existing features. For example, parallel processing requires extension of declarators so as to permit definition of controlled accessing and major compiler modification to support such. Recursion necessitates new memory management policies and routines to implement these new policies.

e. Within the context of FORTRAN, enabling the composition of control structures is a formidable task. The problem lies in the statement orientation of FORTRAN. While the DO structure can be defined over a sequence of statements, no general facility exists for defining composite statements. This can be seen in the logical IF, which can cause conditional execution of only a single statement.

2. The GoTo (G2).

a. Degree of compliance: P

b. The GoTo statement is present in FORTRAN. FORTRAN has numeric labels, switches, label variables and label parameters, but no designational expressions. With the exception of label parameters (the GoTo a label parameter is a special form of the RETURN statement [1, Section 15.8]) the GoTo is limited to explicitly specified program labels at the same scope level [1, Section 11.1, 11.2 and 11.3]. FORTRAN imposes no unncessary costs for the presence of GoTo.

c. The only valid use of label parameters seems to be in conjunction with error handling. If other exception-handling facilities are added to the language (as required by G7), this feature can be easily removed without impacting the power of FORTRAN.

3. Conditional Control (G3).

a. Degree of compliance: P

b. FORTRAN has three types of conditional control. An arithmetic IF jumps to one of three labelled statements, depending on the value of an arithmetic expression. The logical IF will execute or not execute a statement based on the value of a Boolean expression. There is no facility in FORTRAN for defining an "ELSE" clause. The computed GoTo offers a FORTRAN user a CASE-like mechanism.

c. Since [1] was written, four new statements pertinent to this discussion have been added to the draft proposal. These are ELSE, ELSEIF, ENDIF and a block IF. This still leaves the non-partitioned logical IF, however. Its deletion would have a very small effect if the ELSE clause was permitted to be empty.

d. A general form of condition (e.g., Zahn's device) could be added to the language without great difficulty. However, this type of construct is at a much higher level than existing control structures and would not blend naturally with the rest of the language.

4. Iterative Control (G4).

a. Degree of compliance: P

b. FORTRAN's iterative control mechanism permits the termination condition only at the head of the loop. It does not restrict the scope of control variables, and it offers as its only termination condition a fixed number of iterations [1, Sections 11.6.3 and 11.6.4]. Entry to an "inactive" DO-loop is permitted only at the head of the loop but, because of the presence of label variables, this restriction is difficult to enforce. It is possible to pass values out of a loop, but the loop control variable itself has an unambiguous value [1, Section 11.6.7].

c. FORTRAN's deficiency of iterative control structures reflects its relatively early design in terms of the development of structured programming theory. The addition of more modern and general control structures requires a moderate amount of effort in both design and implementation. A general loop-while statement analogous to the one proposed by Knuth [12] would bring FORTRAN into greater compliance with this requirement.

d. Limiting the scope of the loop control variable in the DO statement would constitute a relatively minor change in the language specification. It could be accomplished simply by stating that use as a control variable causes an identifier to become undefined after loop exit. Recognition of references to undefined variables has already been discussed in connection with requirement D3.

5. Routines (G5).

a. Degree of compliance: F

b. FORTRAN explicitly forbids recursion: "A subprogram must not reference itself, either directly or indirectly" [1, Section 15.6.2.2].

c. Adding recursion to FORTRAN requires a minor modification from the language design point of view but extensive support within the compiler. Since FORTRAN is not a block-structured language the issues of restricting the lexical embedding of routines in recursive procedures do not apply. The primary difficulty with introduction of

recursion relates to the requirement for dynamic memory management, which it necessitates. Each new activation of such a routine requires memory to be allocated from some space pool for all local variables. Even if the suggested limitations are placed on recursive procedures, the ability to have mutually recursive routines can cause complex and time-consuming memory management to occur.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. FORTRAN has no facilities for parallel processing.

c. The scope of language modifications needed to support parallel processing would be extremely large. In addition, the effort required during implementation would be great, especially if no operating system support may be assumed.

7. Exception Handling (G7).

a. Degree of compliance: F

b. FORTRAN provides the user no capabilities for retaining control in the event of any non-I/O exception, nor is any standard action specified for such exceptions. The I/O exceptions for which a user retains control are largely implementation-dependent [1, Section 12.6.11].

c. The first step to compliance with G7, is the definition of all possible (or recoverable) error conditions. This would require great effort if total generality is desired. The introduction of language features for specifying the error-handling routines would be much simpler; however, they could necessitate extensive compiler and run-time support. In the absence of hardware detection of these exception conditions, an excessive run-time overhead would be incurred.

8. Synchronization and Real Time (G8).

a. Degree of compliance: F

b. The only facility which FORTRAN contains for causing delay on a program path is the PAUSE statement. This simply provides the ability to stop execution until some undefined

operator intervention occurs. This in no way satisfies the expansive requirements of G8.

c. The PAUSE statement could be expanded to include suspension of execution for a fixed amount of time or until some external event (other than operator intervention) occurs. However, the ease with which this could be implemented greatly depends on the system on which the compiler is meant to run. In the absence of system support, this would entail a large amount of work.

d. Assigning relative priorities among parallel control paths and permitting asynchronous hardware interrupts to be treated as any other exception situation are actually added requirements of G6 and G7, respectively. The difficulty of implementation is dependent on how the modifications suggested in these sections are accomplished and the underlying system configuration.

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. FORTRAN has a "semi-fixed" format: positions 1-5 of a line or card are reserved for a statement label if any is to appear; position 6 is a continuation denoting if the current line or card is a continuation of the previous line or card; positions beyond 72 are ignored; and positions 7-72 contain the FORTRAN statement in a free form. Mnemonically significant identifiers of six or fewer alphanumeric characters may be used. Since blanks are ignored in FORTRAN, lexical analysis is relatively complex. FORTRAN has some context-dependencies and is not easily parsed; on the other hand, there are a number of efficient ad hoc techniques directed especially at parsing FORTRAN so that FORTRAN compilers are generally quite fast. FORTRAN does not allow abbreviations of keywords or identifiers. The FORTRAN assignment statement and FORTRAN expressions are based on conventional forms. The positional significance of DO-statement components and arithmetic IF-statement jump addresses result in a lack of clarity. Because in FORTRAN blanks are not significant and keywords are not reserved, some programmer errors can produce completely unintended but still valid statements (e.g., "DO 315 I=1, 10" executes the DO-loop ten times, whereas "DO 315 I=1.10" assigns 1.10 to a variable DO315I).

c. The changes necessary to permit free format coding of FORTRAN are relatively minor in terms of language design. The additional compiler code for interpreting free-format programming, however, would be reasonably large. Also, from a historical point of view this would constitute a major change in design.

d. Regarding readability, the arithmetic IF can be deleted from the language, since this statement has now been superseded by new conditional constructions. Probably the only reason this has not been done in the new draft proposal is that it would limit upwards compatibility. One should also note here that despite FORTRAN's somewhat unclear statements (e.g., DO) the language is so widely used that most programmers are familiar with their meaning.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. The FORTRAN user has no facilities for modifying any aspect of the source language syntax.

3. Source Character Set (H3).

a. Degree of compliance: T

b. The FORTRAN character set consists of (1) the Alpha characters A-Z, (2) the Numeric characters Ø-9, and (3) the special characters blank = + - * / () , . $ ' : [1, Section 3.1], all of which appear in the USASCII 64 character subset.

4. Identifiers and Literals (H4).

a. Degree of compliance: PT

b. FORTRAN provides formation rules for identifiers and literals. Except for those internal to character string literals, FORTRAN ignores the presence of blanks; thus blanks may be used as break characters for identifiers and numeric literals. The maximum number of characters allowed to compose an identifier is six. FORTRAN does not require separate quoting of each line of a long literal.

c. Requiring separate quoting of each line of a long literal is a trivial change to the language.

5. Lexical Units and Lines (H5).

a. Degree of compliance: F

b. FORTRAN allows the continuation of lexical units across lines. There is no way to include end-of-line in literal-strings.

c. The inclusion of an end-of-line notation within literals implies only a very minor change to the language. For consistency with input-output formatting a slash (/) would probably be best. This would necessitate another notation (possibly a double slash) when one wishes to actually include this symbol. Restricting lexical units to a single line requires similarly trivial changes.

6. Keywords (H6).

a. Degree of compliance: P

b. FORTRAN keywords are not reserved, are few in number, and are relatively informative. Since they are not reserved, any keyword may be used as an identifier.

c. Reserving all keywords constitutes an extremely minor language modification. Implementation would be made easier since parsing would be simplified and error recovery could be improved. The ability to define identifiers which are also keywords is seldom used and consititutes poor programming.

7. Comment Conventions (H7).

a. Degree of compliance: P

b. FORTRAN has a single uniform comment convention. Comments may be easily distinguished from code, permit any combination of characters to appear, terminate automatically at end of line, and are introduced with a single language character. FORTRAN comments are introduced with a "C" or an "*" in position 1 of a source line or card [1, Section 3.2.1]. This convention prohibits some reformatting of programs. A major drawback is that a comment may not appear on the same line as the statement to which the comment applies.

c. Required here is a second comment convention which is fully bracketed and may appear internal to statements. The additional work needed in both design and implementation is extremely small.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. Parentheses must be balanced in FORTRAN programs. Every program unit requires an END, and every DO-loop requires a labelled end-of-loop statement.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. FORTRAN has one referent notation which is valid for function calls [1, Section 15.2.2] and data reference [1, Section 5.3]. The maximum number of dimensions of an array is seven but the maximum number of arguments of a function is not explicitly specified. The fact that function calls cannot appear on the left side of assignment statement is in conflict with the uniform referent requirement.

c. The limit on the number of possible array dimensions seems reasonable and is required to comply with I6.

d. Incorporation of a facility for permitting assignment to functions requires major work in both design and implementation.

10. Consistency of Meaning (H10).

a. Degree of compliance: T

b. FORTRAN has a consistent notation. For instance, "=" is the assignment symbol and ".EQ." is the comparison operator. FORTRAN conventions do imply different interpretations for parenthesized arguments, but that is consistent with the defined syntax and semantics for expressions appearing in argument lists [1, Section 15.9.3].

Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

   a. Degree of compliance: F

   b. FORTRAN fails to satisfy this requirement; in fact, the standard asserts that whether illegal programs are diagnosed is implementation dependent [1, Section 2Ø.1]:

> What this standard calls a "processor" is any mechanism that can carry out the actions of a program. Commonly this may be any of these:
> (1) The combined actions of a computer (hardware), its operating system, a compiler, and a loader.
> (2) An interpeter.
> (3) The mind of a human, perhaps with the help of paper and pencil.
> When you read this standard, it is important to keep its point of view in mind. The standard is written from the point of view of a programmer using the language, and not from the point of view of the implementation of a processor. This point of view affects the way you must interpret the standard. For example, in section 3.3 the assertion is made:
> ".... a statement must not contain more than 132Ø characters."
> This means that if a programmer writes a longer statement, then his program is not standard-conforming. Therefore, it will get different treatment on different processors. Some processors will accept the program, and some will not. Some may even seemingly accept the program but process it incorrectly! The assertion means that standard-conforming processors must accept statements up to 132Ø characters long. That is the only inference about a standard-conforming processor that can be made from the assertion.
> The assertion does not mean that a standard-conforming processor is prohibited from accepting longer statements. Accepting longer statements would be an extension.
> The assertion does not mean that a

standard-conforming processor must diagnose statements longer than 1320 characters, although it may do so.
In some places, explicit prohibitions or restrictions are stated, such as the above statement length restriction. Such prohibitions are on what programmers can write in standard-conforming programs. As such they have no more weight in the standard than an omitted feature. For example, there is no mention anywhere in the standard of double precision integers. Because it is omitted, programmers may not use this feature in standard-conforming programs. A standard-conforming processor may or may not provide it or diagnose its use. Thus, an explicit prohibition (such as statements longer than 1320 characters) and an omission (such as double precision integers) are equivalent to this standard.

c. As can be seen from paragraph b, the design philosophy of FORTRAN is in direct opposition to this requirement. For this reason, compliance would entail fairly extensive modification of [1]. The limits set on standard-conforming programs need to be extended to include the translator and the action to be taken when a non-standard form is encountered. This would greatly limit the flexibility of implementation and necessitate more development effort before any compiler could be judged standard-conforming. It also would set a lower limit on the complexity of any compiler. This would increase the minimum size of a FORTRAN translator and thereby prohibit its use on certain machines.

2. Object Representation Specification Optional (I2).

a. Degree of compliance: F

b. The FORTRAN language has no provisions for specifying the object representation of data or programs.

c. The addition of object representation specification facilities would constitute a major redesign of the language and greatly complicate implementation. The impact on existing facilities would be both pervasive and complex. For example, the COMMON and EQUIVALENCE statements would

need to be either restricted to associating variables of identical object representation or drastically changed in nature (possibly totally removed from the language). New rules for parameter passing and operations between variables of different object storage would have to be defined and adhered to by implementations.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. FORTRAN contains no compile-time variables and has no facilities for their definition.

c. This requirement necessitates relatively minor changes to the language since the addition of compile-time variables does not impact present facilities. The only possible interaction would occur when a programmer attempts to define an identifier with the same name as a compile-time variable. This possible ambiguity can easily be removed by the establishment of a standard interpretation.

d. The extent of needed compile-time support of this feature depends upon the number and complexity of inquiries allowed. If the compiler is to maintain an extremely detailed model of the object machine, the task of both designing and implementing these facilities will greatly increase in difficulty. In addition, in a quickly evolving environment frequent compiler updates would be required. However, a relatively simple model could be developed to include the system parameters required by I3, and it is to such a model that the comments in c apply.

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. FORTRAN contains no conditional compilation facilities.

c. A simple conditional compilation facility could be added to FORTRAN fairly easily. Real difficulties can be encountered, however, if this capability is made overly general and frequently used. For example, if the compilation becomes subtly linked to a particular machine configuration, then small changes in the system could cause

new and hard-to-locate bugs. This is especially true when a program is made installation-dependent (as opposed to machine dependent).

5. Simple Base Language (I5)

a. Degree of compliance: PT

b. The concept of a "base" or "kernel" language is inapplicable to FORTRAN, since FORTRAN is very weak in definitional facilities. Thus the entire language must be considered as the base. Seen in this light, FORTRAN is fairly successful in satisfying I5. The language is relatively simple, and there are few duplicated facilities (examples of the latter are the WRITE and PRINT statements, and logical and arithmetic IF statements). FORTRAN object code can be efficient, and its degree of safety depends on the implementation and whether "dangerous" features (such as EQUIVALENCE) are used. Program understandability depends on whether (and how) features such as label assignment and the COMMON statement are used.

c. The scope of modifications required here depends upon the changes made to the language in conjunction with other requirements. For example, if extension facilities are added, the "base" would need to be extended to include a facility by which these could be defined.

d. As it stands, FORTRAN requires only minor changes to bring it into full compliance. These would be primarily in the direction of pruning the language of redundant features (e.g., the arithmetic IF and statment functions).

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. FORTRAN limits explicitly the number of array dimensions and the length of identifiers but does not limit the levels of nested parenthesis or the number of identifiers.

c. The addition of all required limitations to the language definition is straightforward. The problems lie in making the distinction between machine and translator limits. FORTRAN is probably used on more different machines

than any other language. To expect the FORTRAN used on the CDC STAR to have no more identifiers than some maximum determined for a PDP-8 seems unreasonable.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. FORTRAN has no restrictions which are a result of a specific object machine implementation.

Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: PT

b. FORTRAN does not impose run-time costs for unneeded or unused generality, and compilers for FORTRAN should be capable of producing efficient code for all programs. It may be noted that there has been an exception to this with past versions of FORTRAN with respect to character processing. The definition of character strings given in [1] should allow better code to be produced for programs processing character data. The FORTRAN language specification does not state how features in the language are to be implemented; however, the use of libraries is common and thus only the capabilities used need be available at run time.

c. A formal definition of library facilities could be easily added to [1]. This would, if done properly, bring FORTRAN into full compliance with this requirement. (See our above comments concerning F4.)

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: P

b. FORTRAN has attempted to meet this requirement by leaving order of evaluation unspecified but restricting the side effects permitted in FUNCTIONs [1, Section 6.6.2]:

> If a statement contains more than one function reference, a processor may evaluate the functions in any order, except for a logical IF statement and a function argument list containing function references...
>
> In a statement that contains more than one function reference, the value provided by each function reference must be independent of the order chosen by the processor for evaluation of the function references.

However, since enforcement of the side-effect restrictions is outside the scope of the language standard, in practice different orders of evaluation may yield different results. In addition, side effects not prohibited by the above rules

can cause problems. For example, assume that F is a function which assigns its argument to the global (COMMON) variable I and then returns this value. Then the expression F(1) + F(2) will yield the same result (3) whichever invocation is evaluated first, but the value of I on completion of the evaluation is dependent on the order in which F(1) and F(2) are performed.

c. The enforcement of side-effect restrictions requires considerable additions to the compiler. Instead of the needed global program analysis, many common FORTRAN optimizations would be forced to be removed from standard-conforming compilers. This could have drastic effects on program efficiency. (These issues were also reviewed in connection with requirement C1.)

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. FORTRAN permits external procedures to be coded in any language (including machine language) [1, Section 15.6]; however, the responsibility for proper parameter handling is the programmer's. This is not, in fact, as great a difficulty as it sounds, due to FORTRAN's extremely simple parameter passing mechanism. Additionally, FORTRAN does not allow these (or any other) routines to be declared open, thereby limiting their utility in optimization. Since conditional compilation is not available in FORTRAN (although required by I4) these subprograms cannot be encapsulated as suggested.

c. The problems associated with allowing open procedures are small and are discussed below in connection with J5. A conditional compilation capability is fairly easy to implement. (See our discussion of I4 above.) The subroutine mechanism provides most of the needed encapsulation characteristics. Taken as a whole, the few changes required for full compliance are small and implementation would not be greatly complicated.

4. Object Representation Specifications (J4).

a. Degree of compliance: F

b. FORTRAN provides no facilities for specifying object representations of data, nor is there, within the standard, any ability to specify space/time tradeoffs.

c. The modifications needed to support object representation specification are reviewed in connection with I2. The introduction of optimization directives requires only minor changes in the language definition but possible extensive support from the compiler. Due to the simplicity of FORTRAN and the specification within [1] of required storage characteristics for all variables, very few space/time tradeoffs may be made by the translator. However, if the language is modified to permit greater compliance with the Tinman, many such situations could arise.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: F

b. The FORTRAN user has no ability to specify whether an open or a closed routine is to be used.

c. This requirement necessitates a trivial change to the language definition (i.e., adding the ability to declare subroutines opened or closed). Some changes to the implementation are necessary; however, these would be fairly minor.

AD-A037 639 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
CANDIDATE LANGUAGES EVALUATION REPORT.(U)
JAN 77 B M BROSGOL, R E HARTMAN, J R NESTOR DAHC26-76-C-0006
UNCLASSIFIED IR-217 USACSC-AT-76-11 NL

4 OF 6
ADA037639

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

   a. Degree of compliance: F

   b. While no explicit reference to the existence of an operating system appears in [1], an underlying assumption of one's availability is made.

   c. There are many aspects of FORTRAN which in the absence of an OS would require extensive run-time support. The I/O statements in FORTRAN require a file and device management system. Inquiry routines must be supported to return information about the characteristics of these I/O objects. A direct (random) access mechanism must exist. Implementation of a compiler which alone satisfies all the requirements of the present language would be a large task. In addition, if facilities exist to meet the other Tinman directives (e.g., parallel processing, exception and interrupt handling), the job would become even more difficult.

2. Program Assembly (K2).

   a. Degree of compliance: U

   b. The language definition [1] neither prohibits nor requires a compiler to have the capability to support integration of separately written modules. The EXTERNAL declarator implies a separate definition facility, but this facility is not described.

   c. Many installations support the required capability. No feature of the language prohibits or even greatly complicates such a facility. The EXTERNAL statement can be used in conjunction with separately written modules and provides most of the specification abilities required. However, if complete interface checking is to be easily accomplished then this statement should be expanded to include specification of the number and types of parameters. This requires only minor language redesign and small additions to the compiler.

3. Software Development Tools (K3).

a. Degree of compliance: U

b. The family of tools available for a given language does not form part of its definition.

c. Due to FORTRAN's widespread use and relative simplicity many software tools have been developed for it. To supplement FORTRAN's relatively weak structured control facilities many preprocessors have been developed for use with it (e.g., see [13] which catalogs over 50 of them). In addition, many linkers, loaders and debuggers exist for the language.

4. Translator Options (K4).

a. Degree of compliance: U

b. This requirement consists of a set of suggestions for translator options, and explicitly prohibits the language definition from defining any given set of options. [1] makes no reference to possible compiler options.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: F

b. FORTRAN contains no facility (except for comments) for permitting inclusion of assertions, assumptions, axiomatic definitions of data type, debugging specifications, or units of measure.

c. The introduction of new commenting form(s) for these purposes would be trivial.

Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

a. Degree of compliance: F

b. [1] states that "a standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships." Therefore, FORTRAN totally fails to meet the requirement (i.e., it permits virtually unlimited superset implementations).

c. Removing permission for superset implementations from [1] and replacing it by an explicit prohibition requires rewriting of only a small portion of the defining document. This, however, precludes the use of such a mechanism for realizing the needed modifications already discussed. If such a mechanism is not used, it implies a lack of upwards compatibility. This would create a great uproar due to the deep entrenchment of present FORTRAN.

2. No Subset Implementations (L2).

a. Degree of compliance: F

b. Approximately half of [1] is devoted to the definition of a FORTRAN subset.

c. The FORTRAN subset was defined to encourage the use of the language in contexts where the compiler for the full language would require too great a development effort or would not fit on the current computer. If the subset definition is removed from [1] (a simple task), FORTRAN could not be used in these situations. The extension of a subset compiler to handle the entire language would be a large job.

3. Low-Cost Translation (L3).

a. Degree of compliance: T

b. To the extent that language design can affect translation costs, FORTRAN encourages low-cost compilation. The language is relatively simple, and few real requirements are placed on what the compiler must do.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This requirement is directed at translator design and has small impact on language definition. It should be noted, however, that FORTRAN is a widely used HOL and has been implemented on more machines than any other language.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement applies only to translator design.

6. Translator Checking Required (L6).

a. Degree of compliance: F

b. [1] is an extremely weak standard. Any actions, including none, taken by a translator when supplied with an incorrect (not standard-conforming) program, are valid. This implies that violations of the language syntax, type compatibility rules, or semantics restrictions can go unreported.

c. The modifications to [1] which are required to make it a strong standard are relatively small. The definition of a standard-conforming translator must be changed. This could somewhat complicate implementation of small and fast compilers, since at present they can ignore possible standards violations.

7. Diagnostic Messages (L7).

a. Degree of compliance: FU

b. No set of error and warning situations are suggested in [1]. This being the only language-design-oriented requirement of L7, FORTRAN fails to comply.

c. The definition of error situations and messages would be a fairly small task and could be included in [1]. The implications on compiler implementation of forcing error reporting have been discussed in connection with requirement L6.

8. Translator Internal Structure (L8).

a. Degree of compliance: T

b. The FORTRAN language definition does not dictate the characteristics of translator implementations. The philosophy of [1] can be seen in the following statement: "What this standard calls a 'processor' is any mechanism that can carry out the actions of a program."

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. The language in which a translator is written is not the province of the language definition.

c. The difficulties with writing a compiler in FORTRAN stem from its inability to easily define the necessary data structures and to interface cleanly and efficiently with the object machine.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: T

b. FORTRAN, being one of the earliest HOLs, has been thoroughly tested in practical applications. The new constructs and facilities added by the draft proposal [1] are all well within the current state of the art.

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. The draft FORTRAN proposal was written with emphasis upon readability. The document is directed towards use by programmers and presents only an informal English specification of language semantics. Ambiguities have been avoided, however, and a formal "railroad track" definition of syntax is presented as an appendix to the report. The major drawback of [1] is that it defines only the operation of a standard-conforming program and allows the reader to infer the translator requirements. Formal specification of the language (e.g., in VDL) does not form part of the defining document. Because of its idiosyncracies and many storage-oriented constructs, FORTRAN is not readily amenable to formal definition.

3. Language Documentation Required (M3).

a. Degree of compliance: P

b. FORTRAN syntax is presented for each statement in an easily-understood metalanguage, and the corresponding semantics are given in English. The language is fairly easy to learn, and the defining document is directed at potential users. The action of any legal program cannot be determined from the program and the language definition due to implementation-dependent I/O statements, character collating sequence, and variable range and precisions.

c. Discussion of these implementation-dependent language features occurs earlier in connection with requirements B10 for I/O, A5 for collating sequences, and A3 and A4 for precision and range.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. Management of FORTRAN is outside the scope of the defining document. One can, however, foresee trouble in any attempt to enforce DoD standards on this language. Due to the widespread use and large industrial investment in FORTRAN, any attempt to control the language and its translators, especially if drastic modifications are planned, will meet resistance.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. Identification of support agents responsible for maintaining the translators and language aids is not within the province of the language definition. Support on a DoD-wise basis should be unaffected by the language chosen.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. Maintenance of common libraries is outside the province of language definition. Some difficulty will arise in this area due to the proliferation of already-existing FORTRAN libraries; however, DoD-wide control and support should be realizable. The comments made above in connection with requirement M4 apply equally well to any attempt to fully control FORTRAN libraries.

Section XV. CONCLUSIONS REGARDING FORTRAN

1. Conflicts between Objectives of FORTRAN and Tinman.

a. Of major consideration in the drafting of [1] was compatibility with the earlier FORTRAN standard. This was inevitable, due to the large investment already existing in FORTRAN software. It implies, however, that the language is still locked into the original design concepts. The last FORTRAN standard, ANS X3.9-1966, was published in 1966, and the theory of language design has evolved greatly since. An increasing emphasis on readability, reliability, and abstraction techniques has occurred, and this new emphasis is visible to only a very limited extent in [1].

b. In FORTRAN's design, portability is secondary to availability (i.e., ease of implementation). Evidence of this exists in the weakness with which limitations are placed on translators for the language. In the case of incorrect programs, the translator is empowered to take whatever actions (including none) that it sees fit. This means that possible semantic violations need not even be checked. To further the use of FORTRAN in differing environments, the standard (1) permits language extensions, (2) suggests subset implementations, and (3) leaves undecided many issues that could possibly complicate implementation (e.g., relative ordering of characters and digits, character vs. non-character storage requirements).

c. The Tinman, on the other hand, reflects the present trend in language design. The major emphasis of that document is upon program reliability and readability and upon language power. The goal of efficiency is seen to be achievable through comprehensive user specification facilities and intelligent compilation as opposed to language simplicity.

2. Summary of Major Areas of Conflict Between FORTRAN and the Tinman.

a. Data and Types. An indication of the differing viewpoints can be seen in the limited facilities FORTRAN offers in this area. FORTRAN contains no fixed-point data type besides integers, no record definition facilities, limited and implementation-dependent precision specification and only static memory allocation (e.g., compile-time

determined array bounds). Additionally, while type-compatibility for actual and dummy parameters is required, the translator may optionally omit type checking. Also, the COMMON and EQUIVALENCE statements effectively disable type checking by permitting free union.

b. Operations. The primary conflicts between FORTRAN and the Tinman in this area are the language's failure to support enumeration types, power sets, range specifications, and operations on data aggregates (i.e., arrays). In addition, FORTRAN supports mixed-mode arithmetic with implicit type conversions and only bitwise equality for floating-point numbers.

c. Expressions and Parameters. In this area, conflicts stem from FORTRAN's only partially defined order of evaluation, its lack of dimensionality constraints on array parameters, and its failure to support generic procedures and variable numbers of parameters.

d. Variables, Literals, and Constants. FORTRAN's lack of step size and range specifiers, separate variable initialization and declaration statements, and lack of pointer variables are its major deficiencies in this area.

e. Definition Facilities. This is the area where FORTRAN deviates most from the Tinman's requirements. There are no facilities for user definition of data types. The only data definition available is the array. For this reason the issues of encapsulated type definitions and operator extensions do not even apply. In addition, FORTRAN permits free union through the EQUIVALENCE statement.

f. Scopes and Libraries. While [1] does not explicitly define library and compool facilities, their inclusion within the language is relatively simple. The EXTERNAL declarator implies the existence of such a feature.

g. Control Structures. FORTRAN provides only a limited control structure. The degree of effort necessary to force FORTRAN into compliance in this area seems prohibitive. The language contains no parallel processing, no exception- or interrupt-handling facilities, no recursion, and no facilities for composing control structures. In addition, the logical IF is not fully partitioned.

h. Syntax and Comment Conventions. FORTRAN is a semi-fixed format language. No separate quoting is required for multi-line literals, and lexical units may continue across lines. Keywords are not reserved and only full-line comments are available.

i. Defaults, Conditional Compilation, and Language Restrictions. FORTRAN seriously conflicts with the Tinman philosophy of no defaults. The most obvious default is that relating the first letter of a variable with a given type. Possibly more crucial, howevever, is the implementation-dependent range and precision associated with numeric variables and the only partially-defined character collating sequence. In addition, FORTRAN lacks compile-time variables and facilities for conditional compilation and object representation specification.

j. Efficient Object Representations and Machine Dependencies. Efficiency is achieved in FORTRAN through simple language constructs and permissive optimization policies. This results in optimizations which can change the effect of the computations. The language includes no space/time tradeoff directives and no open procedure specifiers.

k. Program Environment. The program environment requirements are for the most part beyond the province of language definition. To the extent that [1] impacts on these requirements, only small differences exist.

l. Translators. A basic disagreement of philosophies exists in this area. The Tinman requires a strict language standard allowing no superset of subset implementations, whereas [1] defines a compatible subset and permits unlimited supersetting.

m. Language Definition, Standards and Control. In terms of language definition, FORTRAN fares well. [1] is extremely readable, although arbitrary implementation dependencies led to only partial semantic definition of several language features. Control of FORTRAN can be seen to be difficult. The widespread use and large amounts of software already existing in the language led to many groups with vested interests in controlling the language.

3. Unnecessary Features of FORTRAN

a. Several features of FORTRAN are neither prohibited nor required by the Tinman. The arithmetic IF, ASSIGN, assigned GOTO and statement function statements are all relics of earlier versions of FORTRAN which fall into this category. The mechanism supplied for creating label variables and jumping to locations thereby specified conflicts with the Tinman philosophy of reliability. Since label variables are declared to be of type integer, this entails a violation of type rules. In addition, the availability of label variables prohibits compile-time enforcement of control transfer rules (e.g., jumping into the middle of a DO loop). The arithmetic IF and statement function statements are now redundant due to the introduction of the logical IF and FUNCTION subprograms, respectively.

b. The ENTRY statement which has been introduced into FORTRAN by the new draft proposal is another feature not impacted by Tinman requirements. However, it provides a unique facility in FORTRAN and should not be removed. Along with the SAVE declarator it permits limited access data structures to be defined.

4. Recommendations concerning FORTRAN.

a. We began this section by discussing the basic differences of philosophy between FORTRAN and the Tinman. Throughout this chapter, we have seen the many fundamental modifications that are needed for FORTRAN's compliance with the Tinman. In addition, this section has emphasized the expected resistance to any major changes in the language. For these reasons, it is felt that the choice of FORTRAN as a base for development of a Tinman-like language would be ill-advised.

b. The potential advantages of selecting a language already in widespread use are obvious (availability of programmers, translators, support tools and documentation). However, the changes required in FORTRAN are of such a global nature as to eliminate these potential gains. The necessary modifications would drastically alter the language necessitating retraining of programmers, and rewriting of existing software and documentation.

c. In summary, FORTRAN as proposed in [1] is incompatible with the Tinman, and modification to achieve compliance would be prohibitive and destroy the existing "flavor" of the language.

CHAPTER 7

COBOL EVALUATION

Section I. LANGUAGE SUMMARY

1. Lexical Properties.

COBOL is primarily card-oriented rather than free-format, as illustrated by the rules concerning the placement of continuation lines and division and section headers. Spaces are significant in separating lexical units. Identifiers may be up to thirty characters long, and the hyphen serves as a break character. There are a large number of reserved words (about 300), including some with alternate spellings and abbreviations. An asterisk in the continuation indicator area is used to denote comment lines.

2. Data Types.

a. The data types available in COBOL may be categorized according to Figure 6.

b. Except for Index data, each elementary data-item is considered to be a fixed-length string of characters of various kinds. A numeric item may contain any of the digits ('0' through '9') with a possible sign. An alphabetic item may contain any of the letters or the space character. An alphanumeric item may contain any character in the computer's character set. Numeric, alphabetic, and alphanumeric are called "classes" in COBOL and correspond roughly to data types; however, run-time checks must be made to guarantee that data declared as numeric or alphabetic in fact do not contain improper characters. It should be pointed out that the central role of the character string in COBOL's data scheme is not surprising, in view of the heavy orientation of that language toward the I/O requirements of business data processing.

c. As shown in Figure 6, numeric data are of two varieties. The condition data facility provides a limited form of enumeration type, allowing a symbolic name to be used (but only in conditional statements) in place of a test involving an explicit numeric value. We use the term "standard" numeric to refer to numeric items which are not defined as condition data.

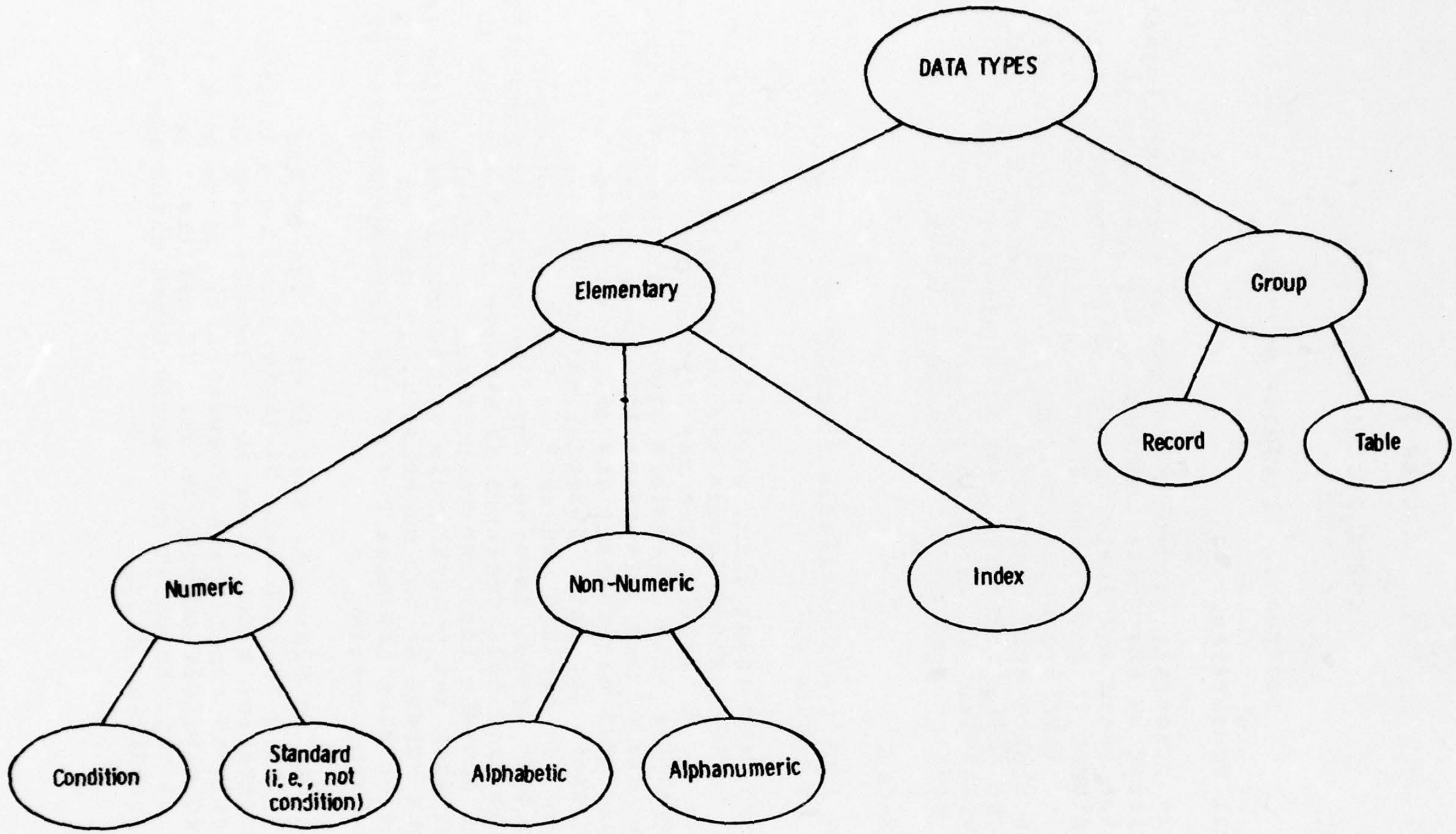

Figure 6. Data Types in COBOL

d. An Index data-item has special use in table handling. It is provided for efficient subscripting on the object machine.

e. Group data-items may be records (hierarchically composed structures) or tables (one-, two-, or three-dimensional arrays). Independent of the classes of items, a group is regarded as a string of alphanumeric characters for the purpose of using it as a complete entity (e.g., in MOVEs).

f. Overlaying of storage is allowed in COBOL via the REDEFINES and RENAMES clauses.

3. Procedures.

COBOL's subroutine facility is quite weak. "Function" subroutines (i.e., routines which return values) are not provided. In order to define a routine which takes parameters, that routine must be embodied in the PROCEDURE DIVISION of a separate program; moreover, the only kind of parameter passing allowed is "by reference," and no type checking is performed. Within a single program, the only kind of routine which may be defined is one which takes no parameters and which has access to all the data in the DATA DIVISION.

4. Statements.

a. Arithmetic Statements. The ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements supply COBOL's arithmetic capabilities.

b. Assignment. Assignment in COBOL can be carried out via any of the arithmetic statements or through the MOVE statement. Implicit conversions (from display to computational format or vice versa) are performed.

c. GoTo. COBOL's GO TO statement allows branching to any paragraph (or section) in the PROCEDURE DIVISION. Dynamic modification of the target of a GOTO is permitted via the ALTER statement.

d. Conditional Statements. COBOL provides an IF statement, with both THEN and ELSE clauses mandatory (except when the statement is at the end of a sentence, in which

case the ELSE clause may be omitted). The GO TO ... DEPENDING statement serves as a restricted form of the "case" construct.

e. Iteration. The versatile PERFORM statement serves both to invoke a procedure and to carry out iterated loops. In the latter capacity, the PERFORM statement has options permitting a fixed number of iterations as well as loop repetition until a Boolean condition (tested at the beginning of each iteration) becomes true. The specification of a loop with a test for completion in the middle is awkward.

f. Miscellaneous. A variety of special purpose statements is supplied by COBOL. Examples are the Nucleus Module's INSPECT statement, which permits tallying and/or replacement of characters in a data item, and the Sort-Merge module's SORT and MERGE statements.

5. Storage Allocation.

COBOL provides for static allocation of data; i.e., the sizes of all records and data-items are known at compile-time. (With "varying length" tables, the maximum size is known at compile-time.) The only exception to this occurs when an externally defined routine is invoked (i.e., using the Inter-Program Communication Module); in such a case the storage for the called routine is (implicitly) allocated dynamically, and may be (explicitly) deallocated under program control.

6. Process Scheduling.

COBOL contains no facilities for process scheduling.

7. Files and I/O.

The variety of I/O facilities provided by COBOL is the language's main strength. The following paragraphs, from [8, pp. XIV-38, 39], summarize the mechanisms available.

a. Sequential Organization. "A file whose organization is sequential can only be accessed in the sequential mode. Records in such a file can be accessed in the sequence established as a result of writing the records to the file. A sequential mass storage file may be used for input and

output at the same time. One file maintenance method made possible by this facility is to read a record, process it, and, if it is updated, return it, modified, to its previous position."

b. Relative Organization.

(1) "A file whose organization is relative can be accessed either sequentialy, dynamically, or randomly. Sequential access provides the same results as if the file were organized sequentially. Random access allows the sequence in which the records are accessed to be controlled by the programmer. Each record in a relative file is identified by an integer value greater than zero which specifies the record's logical ordinal position in the file. The desired record is accessed by placing its relative record number in a Relative Key data item. Such a file may be thought of as a serial string of areas, each capable of holding a logical record. Each of these areas is denominated by a relative record number. Records are stored and retrieved based on this number. For example, the tenth record is the one addressed by relative record number 10 and is in the tenth record area, whether or not records have been written in the first through the ninth record areas."

(2) "In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements."

c. Indexed Organization.

(1) "A file whose organization is indexed can be accessed either sequentially, dynamically, or randomly. Sequential access provides access to the records in the ascending order of the record key values. The order of retrieval of records within a set of records having duplicate record key values is the order in which the records were written into the set."

(2) "In the random access mode, the sequence in which

records are accessed is controlled by the programmer. Each record in the file is identified by the value of one or more keys within that record, and the desired record is accessed by placing the value of its record key in a record key data item before accessing the record."

(3) "In the dynamic access mode, the programmer may change at will from the sequential access to random access by using appropriate forms of input-output statements."

8. Exception Handling.

COBOL offers a limited set of features for handling exception conditions arising on arithmetic overflow, I/O, and storage bounds overflow. A user-supplied SIZE ERROR phrase can be included in arithmetic statements to be invoked on overflow (including zero-divide). A FILE STATUS data item, AT END phrase, and USE statement are constructs which support handling of I/O exceptions. An ON OVERFLOW phrase allows user control when storage overflow occurs. Further details concerning these features are supplied below, in connection with the discussion of paragraph G7.

9. Compile-Time Facilities.

a. COBOL's COPY statement provides a limited macro facility, allowing the insertion of source text (e.g., commonly-used file or record descriptions) into a program.

b. The facilities of the Debug Module allow the user to monitor, during program execution, the performance of user-selected procedures and the values of user-specified data items.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: FP

b. COBOL satisfies this requirement only to a limited extent. In fact, the notion of "data type" does not readily apply to COBOL. Nearly all elementary data-items, and group data in certain contexts, are regarded as character strings. Properties which are compile-time checked and part of the data type in other languages require run-time checking (sometimes programmer-supplied) in COBOL. For example, it is possible to declare a data-item with a numeric class (77 TEMP PICTURE 999 VALUE ZERO.), but this does not prevent the item from containing non-numeric data (e.g., as inserted during a READ). If the programmer foresees such a possibility, he must provide a validation routine to ensure that the data match the specified picture. For example:
INSPECT TEMP REPLACING LEADING SPACES BY ZEROES.
IF TEMP NOT NUMERIC GO TO DATA-ERROR.

c. It would be a major change, both to the language and implementation, to bring COBOL into compliance with A1. The effort involved would amount to a redesign of much of the language.

2. Data Types (A2).

a. Degree of compliance: P

b. COBOL supplies most of the types required in A2. Integer and fixed-point (up to 18 digits) are combined in the numeric class in COBOL, and character corresponds to alphanumeric. There is no explicit mention of floating-point in the COBOL document. The discussion of the USAGE clause [8, p. II-35] seems to imply a single computational representation (viz., fixed-point), which would rule out floating-point; however, the description of arithmetic expressions [8, pp. II-39-40) leaves to the implementor the treatment of arithmetic expressions, which appears to allow floating-point as a possible implementation technique. COBOL does not contain an explicit Boolean data type but does allow Boolean operations on relationals to appear in conditional statements. Arrays (up to three dimensions) and records are provided.

c. The addition of floating-point is a minor change to the language but requires a moderate amount of implementation effort. Introducing a Boolean type presents problems because of the special rules which would be required to describe the external format of Boolean data. The specification of arrays and records as type generators would be a major change and require substantial redesign.

3. Precision (A3).

a. Degree of compliance: F

b. Lacking a language-defined floating-point facility, COBOL fails to meet this requirement.

c. As mentioned in connection with A2, the addition of floating point is a minor change to the language and has moderate impact on implementation. Providing global (as opposed to data-item by data-item) precision specification would not be difficult, but it would be in direct conflict with the language's conventions concerning fixed-point data.

4. Fixed Point Numbers (A4).

a. Degree of compliance: T

b. COBOL satisfies this requirement completely via its NUMERIC class data.

5. Character Data (A5).

a. Degree of compliance: P

b. Although COBOL lacks a general facility for defining enumeration types, the language does give the programmer the ability to define character sets in a manner similar to enumeration types. In the OBJECT-COMPUTER paragraph of the ENVIRONMENT-DIVISION, the programmer can specify an alphabet-name to be used as the PROGRAM COLLATING SEQUENCE [8, p. II-6]; the description of the collating sequence (e.g., ASCII, or NATIVE to the object computer, or user-defined) appears in the SPECIAL-NAMES paragraph.

c. The changes needed to enable definition of enumeration types in COBOL are described below in connection with E6. It should be pointed out, though, that the

character data type is unlikely to be cleanly obtained using such a facility. For COBOL many of the problems of defining and using enumeration types are more severe than for other languages, since all data attributes in COBOL are described with PICTURE and USAGE clauses. Finding an appropriate format for representing an enumeration type, descriptions of strings of enumeration types and of literal strings of enumeration types are tasks of great difficulty.

6. Arrays (A6).

a. Degree of compliance: P

b. COBOL partially meets this requirement in its Table Handling module. The number of dimensions is restricted (one, two or three), the value of the lower bound is always 1, and the value of the upper bound is always specified as a literal value. COBOL does provide a limited facility for defining "flexible" arrays -- the phrase "OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-1" [8, p. III-2] -- but the programmer must maintain the current upper bound in data-name-1. An annoyance in COBOL is the inability to define tables at the "top level." If one wishes to define TAB1 as an array of three digit integers, one must embed TAB1 in a record to do so -- e.g.,

```
01 TAB1-RECORD
   02 TAB1 OCCURS 20 TIMES
           PICTURE 9(3).
```

c. Some of the changes required to comply fully with A6 would alter the basic structure of COBOL. The requirement that "The upper subscript bound will be determinable at entry to the array allocation scope" implies block structure which is absent from COBOL. There are no enumeration types available to use as subscripts. Both of these represent major changes to COBOL.

7. Records (A7).

a. Degree of compliance: P

b. COBOL provides several facilities for defining records with alternative structures, but in each the responsibility is placed on the programmer for checking at run-time which alternative applies. Thus the dangers and the possibility for defeating type-checking are inherent in COBOL.

c. Some COBOL features which specify overlaying are:

(1) the definition of multiple records associated with the same file in the FILE-SECTION of the DATA-DIVISION (the overlaying here is implicit and a possible source of programmer error);

(2) the REDEFINES clause in a record description, which allows the same storage to be specified by different data description entries [8, p. II-27];

(3) the RENAMES clause, which allows alternative, possibly overlapping,, groups of elementary items [8, p. II-29];

(4) the SAME AREA clause, which permits the same storage area to be used for records from more than one file (assuming that no two of these files can ever be open simultaneously).

d. As is frequent in COBOL, there are a fairly extensive set of rules and special cases governing the use of these facilities. For example, it is illegal [8, p. II-27] to REDEFINE a storage area whose description includes an OCCURS clause (i.e., a table), but a table is allowed to REDEFINE another storage area. Also, the reference manual does not define the semantics of REDEFINEing a field which includes an INDEX item -- i.e., rules require that the same number of character positions be present in both the overlaying and overlaid data, but the format of INDEX items is implementation-dependent.

e. In order to comply with A7, COBOL would need a different data description structure allowing the programmer to indicate more specifically which portions of records were related and which condition discriminates between alternate forms of a record. To detect any violations in the input data would involve some difficult implementation problems. COBOL completely lacks the CASE-like construct used in other languages which facilitates processing of records with alternate forms. Provision of all necessary features and imposition of the necessary security for meeting this Tinman requirement would be large tasks.

Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. COBOL partially satisifies this requirement. The assignment operation in COBOL is distributed among a variety of verbs -- e.g., the arithmetic verbs with the GIVING option -- but the principal means of assignment is the MOVE statement [8, p. II-74ff]. The following points in connection with the MOVE statement show the differences between COBOL and the Tinman requirement:

(1) One cannot directly MOVE tables in COBOL; instead, a table must be defined subordinate to a non-table group.

(2) It is not permitted to MOVE a numeric edited data item to another numeric edited data item.

(The above points violate the Tinman requirement that "the assignment operation will permit any value of a given type to be assigned to a variable...of that type....")

(3) Lacking encapsulated type definitions, COBOL does not permit user definition of assignment or access operations.

(4) The MOVEing of group data-items ignores the types of the constituents of the groups, since groups are interpreted as alphanumeric independent of their component items.

(5) The MOVE will in general involve implicit conversions (e.g., numeric to numeric edited). Also, see B8 below.

(6) An INDEX data item is not permitted as either the source or the target of a MOVE.

c. An INDEX data item cannot be initialized with a VALUE-clause [8, p. II-11], nor can an INDEX data item be assigned or referenced using the usual COBOL verbs. An INDEX data item may be assigned only with SET [8,

p. III-11ff], which is reserved for this purpose; an INDEX data item may be referenced in relational tests [8, p. III-6] and table references [8, p. I-89ff]; it may not appear in an arithmetic expression [8, p. II-39].

d. As demonstrated by the rules governing assignment and reference, COBOL and the Tinman were designed with entirely different objectives. Although the class and category of a data name may appear analogous to the Tinman "type," the PICTURE also plays a role in that items in the same class and category with different PICTUREs will require conversion during a MOVE or other assignment. (This accounts for some of the behavior above.) To change this characteristic of COBOL would require comprehensive redefinition of the data description and the internal data organization; the product of such a redefinition would not be recognizable as COBOL.

2. Equivalence (B2).

a. Degree of compliance: P

b. The EQUAL relation in COBOL can be used to compare objects for identity, but the rules provided for its use [8, p. II-41ff] have serious conflicts with the Tinman requirement. First, EQUAL is more general than B2 allows: in COBOL, objects of different data types can be compared for equivalence, with implicit conversions taking place. Second, equivalence of group items takes place by (effectively) a bitwise comparison, independent of the data types of the constituents.

c. The comments under paragraph B7 above apply to this Tinman requirement.

3. Relationals (B3).

a. Degree of compliance: P

b. COBOL permits relational operations (GREATER, LESS, NOT GREATER, NOT LESS) in the same contexts as equivalence; the character collating sequence is used to determine the ordering. COBOL lacks a general enumeration types, and it is not possible to inhibit the definition of the ordering relations.

c. See paragraph B6 for the modifications required for the inclusion in COBOL of enumeration types. It is not possible to define unordered sets in COBOL, as all data are represented by character strings which have the implementation dependent (or ASCII or other) ordering. Inclusion of some method for defining unordered sets would be a large change to COBOL and would be outside the spirit of the language.

4. Arithmetic Operations (B4).

a. Degree of compliance: PT

b. COBOL provides two facilities for performing arithmetic: a set of verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE), and a number of operations (+, -, *, /, **) used in the COMPUTE statement. With the exception that division with a real result is not part of the language (but may be left to the implementor), the requirements in B4 are met. Two points pertaining to the DIVIDE statement are worth making here. First, a format is available which computes the remainder of a division. Second, the DIVIDE...INTO format is sometimes counter-intuitive; e.g., DIVIDE TOTAL-AMOUNT INTO 2 GIVING HALF-SHARE-AMOUNT, despite the fact that it reads as though HALF-SHARE-AMOUNT will receive the value TOTAL-AMOUNT/2, in fact will assign the reciprocal of this value to HALF-SHARE-AMOUNT.

c. The scope of changes necessary to add a floating point data type to the language is presented under paragraph A2 above.

5. Truncation and Rounding (B5).

a. Degree of compliance: P

b. COBOL allows implicit truncation of high-order character positions during assignment [8, p. I-86], contrary to B5. If the user specifies a ROUNDED phrase [8, p. II-50], then rounding as opposed to truncation will occur at the low-order end. If high order truncation occurs and the user has supplied an ON SIZE ERROR clause, a user specified statement will be executed [8, p. II-50].

c. An ON SIZE ERROR clause could be required with all arithmetic verbs. This would be an easily defined and

implemented change; however, it would degrade the efficiency of the program and thus conflict with requirement J1.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. COBOL provides operations for "and," "or," and "not," but not "nor" [8, p. II-45ff]. No short-circuit mode of evaluation exists. One interesting feature of COBOL is the provision for abbreviating Boolean expressions. For example, a EQUAL b OR c is short for (a EQUAL b) OR (a EQUAL c). Also, a condition name can be used by itself instead of an explicit test of a condition variable vs. the value of the condition name.

c. As Boolean expressions are allowed only in limited contexts (IF-statements) a requirement that they be evaluated in short-circuit mode would have only a small impact on the language and an implementation.

7. Scalar Operations (B7).

a. Degree of compliance: F

b. The only provision in COBOL for performing scalar operations or assignment on "conformable" aggregate data is the CORRESPONDING phrase [8, p. II-51] for the MOVE, ADD, and SUBTRACT verbs. However, the correspondence defined in COBOL is dependent on the names of the record components and does not permit operations such as adding a numeric item to an array of such items. The MOVE verb in general allows assignment of any two records, independent of their logical structure; the result is an alphanumeric string copy from the source record to the destination.

c. Restricting arithmetic and MOVE operations to identically structured records would not be a large change to COBOL. However, the inclusion of vector and array operations would be a substantial and undesirable extension to COBOL. The change would be large because tables are embedded in records and because the actions of all operators would need to be extended or changed. Such changes would be undesirable because COBOL does not attempt to be an efficient algebraic language and it has quite different goals from those implied by B7.

8. Type Conversion (B8).

a. Degree of compliance: F

b. COBOL fails to meet this requirement; in fact, implicit conversions (from character code to computational and vice versa) are a fundamental aspect of the language (e.g., in comparisons and assignments). No explicit conversion operations are provided.

c. The considerations involved in B8 illustrate again the differences between the objectives of COBOL and the goals of a Tinman-like language. For example, consider the following declarations:

```
    03 DISPLAY-INTEGER PICTURE 999 USAGE IS DISPLAY.
    03 COMPUTATIONAL-INTEGER PICTURE 999
         USAGE IS COMPUTATIONAL.
```

Both variables have a range Ø to 999. The "representational ideal" for both variables is three numeric characters; the announcement USAGE IS COMPUTATIONAL has the purpose of allowing an implementation to select a more efficient representation, if one exists, for computation. Whether to use more than one representation and the representations selected are implementor design decisions.

d. The elimination of implicit conversions would have a major impact upon the language and its implementations.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: PT

b. COBOL partially complies with this requirement. Numeric ranges are implicitly determined by the PICTURE clause (e.g., PICTURE 9(4) denotes the range Ø through 9999), and no explicit conversion operations are required between ranges. The run-time exception on truncation will be generated if the ON SIZE ERROR clause is provided; this is possible for the COMPUTE and other arithmetic verbs, but not for the MOVE statement, which could (but need not) generate a compile-time error.

c. For full compliance with this requirement, the following two minor changes are possible:

(1) The ON SIZE ERROR clause must appear with all arithmetic statements (see paragraph B5 above).

(2) Any MOVE which could cause high order truncation must be announced at compile time (an ON SIZE ERROR could be required for such MOVEs).

10. I/O Operations (B10).

a. Degree of compliance: PT

b. One of the most striking facets of COBOL is its versatility in the I/O area. The format of a COBOL program reflects the orientation toward file handling, with the four separate divisions and the partitioning of the DATA DIVISION into a FILE SECTION and other sections. Also, COBOL very nicely separates the physical description of a file (which includes information such as label handling and blocking factor) from the logical description of the records comprising the file. The diversity of I/O which can be programmed (e.g., sequential, relative and indexed I/O) distinguishes COBOL from most other high-order languages.

c. However, there are several aspects of B10 which are not satisfied in COBOL. One is the ability to assign and reassign devices dynamically; COBOL requires establishing statically the connection between internal and external file names (this is done in the FILE-CONTROL paragraph in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION). Another area in which COBOL fails to satisfy B10 is in that language's inability to permit user definition of I/O operations. The Tinman suggests that the definitional facilities of the language (e.g., for data types and generic procedures) should be powerful enough to permit the specification of unique equipment and their I/O operations. COBOL completely lacks such facilities; thus, any I/O handling must be built into the language.

d. It should also be pointed out that COBOL contains no provision for binary I/O, which is consistent with the fact that COBOL lacks binary data.

e. Full compliance with B10 would require the entire restructuring of the COBOL/operating system interface and the complete redefinition of the FILE SECTION syntax and semantics. These are major changes.

11. Power Set Operations (B11).

a. Degree of compliance: F

b. COBOL completely lacks such operations; moreover, there is no bitstring data type in the language and no way for a user to represent bitstrings efficiently or define operations such as union, intersection, and complement.

c. An appropriate declaration form for power sets would be extremely difficult to devise within the framework of the COBOL DATA DIVISION (similar problems exist for enumeration types; see paragraph E6 below). Because COBOL has no definition capabilities any power set operations would have to be included in some level of the base language.

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: P

b. Strictly speaking, COBOL fails to satisfy the requirements of C1. In the arithmetic expression A/B + (C/D) the parentheses would force the evaluation of the quotient C/D first, followed by the evaluation of A/B, according to the rules on [8, pp. II-39 and II-40]. Thus, if each division would result in overflow the side-effects would not be carried out from left to right.

c. Two points are worth noting in connection with this issue. First, since COBOL does not have value-returning functions in the sense of languages like FORTRAN, ALGOL and PASCAL, there is no possibility for the evaluation of an expression's constituent to result in the performance of I/O or the modification of a variable. Thus, the kinds of side-effects which can occur during expression evaluation are extremely limited -- errors from arithmetic overflow or out-of-bounds subscripts appear to be the only ones possible. However, this means that no matter which order of evaluation is used the result of an expression will always be the same. The language description could be simplified (and better object code produced) if the order of evaluation were left undefined.

d. A second point concerns the apparent confusion in the COBOL standard concerning the parsing of expressions (i.e., the decomposition into constituent sub-expressions) as opposed to the order in which the operands of an operator are evaluated. The failure to distinguish between these concepts is evidenced in the language requirement for parentheses (which should only be used to guide the parsing) to determine order of evaluation. To see that the concepts of expression parsing and evaluation order are independent, consider the expression (A+B)/(C+D). The parentheses indicate that the operands to the division operator are the sum A+B and the sum C+D; such an operator-operand structure is independent of whether A+B is evaluated before or after C+D.

e. The changes needed to bring COBOL into compliance with C1 are minor with respect to the language and moderate with respect to implementation.

2. Operand Structure (C2).

a. Degree of compliance: P

b. COBOL partially satisfies the requirements of C2; areas of divergence from the Tinman are the following.

(1) Unary plus and minus take precedence over exponentiation; in COBOL, -5**2 has the value 25, whereas more commonly (e.g., in FORTRAN and PL/I as well as by standard algebraic conventions) the value would be -(5**2), i.e., -25.

(2) Exponentiation is "left-associative" in COBOL: I**J**K is interpreted as (I**J)**K. Again, standard algebraic conventions dictate a different parsing, viz., I**(J**K).

(3) The ability to abbreviate combined relation conditions in COBOL [8, p. II-47-48], though a possible convenience in writing programs, can make reading difficult. For example, it may not be immediately apparent that "NOT a = b OR c" is equivalent to "(NOT (a = b)) OR (a = c)".

c. It should also be pointed out that the rules given for determining the constituents of an expression are quite awkward; a clean, simple formulation is possible and would be desirable.

d. The changes needed to bring COBOL into compliance with C2 are minor.

3. Expressions Permitted (C3).

a. Degree of compliance: F

b. COBOL fails to meet this requirement. In subscripting, only literals or data-names are permitted; in indexing, only literals, data-names, or data-names offset by a signed literal are allowed [8, pp. I-89-90]. In the PERFORM statement [8, p. II-78], the number of times a procedure is executed, and the lower bounds of the loop control variable, are restricted to identifiers and literals. Similarly, the DISPLAY statement cannot specify expressions [8, p. II-59]. Moreover, there is a somewhat

strange restriction on the form of relation conditions [8, p. II-41]: comparison of two literal values is not allowed.

c. Allowing expressions everywhere both constants and references to variables are permitted would complicate the COBOL language definition and would be a large change to existing implementations. COBOL is not oriented towards expressions and allows general expressions only in COMPUTE statements and relations.

4. Constant Expressions (C4).

a. Degree of compliance: F

b. COBOL contains no provision for evaluating constant expressions before run-time.

c. It is not a major change to specify pre-run-time evaluation of constant expressions. Because COBOL uses fixed-point data, no precision errors will result.

5. Consistent Parameter Rules (C5).

a. Degree of compliance: T

b. Since COBOL only has one kind of parameter (effectively, to a procedure), by default the parameter rules are consistent.

c. Although COBOL satisfies this requirement, that is only because the language is so restricted. Were COBOL modified to improve compliance with other Tinman requirements, this requirement would need to be included as a design goal.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: F

b. The simple parameter-passing mechanism in COBOL is essentially "by reference;" no type checking is performed. The only requirement for a match is that the actual and formal parameters occupy an equal number of character positions [8, p. XII-4]. The interpretation of this last requirement, when the formal and/or actual parameter is a "flexible table," is not stated in the defining document.

c. The satisfaction of this requirement is dependent upon revisions to the data type facility of COBOL to bring it into compliance with A1 and A2. Major changes are needed.

7. Formal Parameter Kinds (C7)

a. Degree of compliance: F

b. COBOL allows only one type of parameter, "by reference." Moreover, there are restrictions on the kinds of data which can be passed as actual parameters; according to Syntax Rule (4) [8, p. XII-5], it is illegal to pass a component from a record in WORKING-STORAGE.

c. COBOL has a very limited procedure capability. Design of a more comprehensive facility would encompass the issues in this Tinman requirement as well as those in Tinman requirements C5, C6, C8, C9, F1, F2, G5, G7 and H9. Major changes would be necessary.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: F

b. COBOL does not allow the omission of the specification of the attributes of formal parameters. As stated on [8, p. XII-4] "Each of the operands in the USING phrase of the Procedure Division header must be defined as a data item in the Linkage Section of the program in which this header occurs, and it must have a 01 or 77 level-number."

c. This requirement could best be met with a macro facility of some sort. COBOL has available already the rudiments of such a macro facility in the 'COPY... REPLACING' construct. Were this facility made somewhat more comprehensive (a moderate change to the language), COBOL could be adapted to meet this requirement.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. COBOL contains no provision for meeting this requirement. Major changes would be needed to bring COBOL into compliance with C9.

Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

   a. Degree of compliance: F

   b. COBOL contains no provision for associating constant values with identifiers, except for the figurative constants (ZERO, SPACES, etc.) built into the language [8, p. I-81].

   c. Such a provision could be added to COBOL without great difficulty.

2. Numeric Literals (D2).

   a. Degree of compliance: T

   b. COBOL has fairly simple rules concerning literals [8, pp. I-80-81]; the two categories (numeric and non-numeric) cover all the built-in data-types. COBOL complies completely with the D2 requirement.

3. Initial Values of Variables (D3).

   a. Degree of compliance: F

   b. COBOL partially fulfills this requirement; the VALUE clause [8, pp. II-36ff] is the vehicle for supplying initialization information for data-items. Unfortunately, the rules governing its use are laden with special cases and exceptions: the VALUE clause cannot initialize some kinds of data (such as INDEXES and records containing flexible tables) and cannot be used practically to initialize other kinds. The following illustrates the impractical application just mentioned: since a record is considered alpha-numeric regardless of the PICTUREs of its constituents, the only legitimate initial value which can be supplied in a VALUE clause is a non-numeric literal or a figurative constant. Initialization of tables is similarly restricted; in order to initialize a table of numbers to its constituent values, one must first define a record giving VALUE clauses to its components, and then REDEFINE this with a table -- a somewhat clumsy and circumlocutive arrangement. It is not possible to specify repetition factors (as found, for example, in FORTRAN's DATA statement).

c. COBOL complies with D3's requirement for no default initial values. In the absence of a VALUE clause, the initial value for the item is undefined.

d. COBOL fails to comply with the requirement that "there will be provision (at user option) for run time testing for initialization."

e. As COBOL has already an initializing capability, the major effort needed to conform to this Tinman requirement involves extending the capability to all data types and making the applications consistent.

f. Runtime testing for initialization is very expensive unless supported by hardware.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. COBOL partially satisifies this requirement via its PICTURE clause [8, pp. II-18 ff]. The reason that COBOL meets D4 only partially is that ranges are restricted solely via the specification of decimal places. Thus data items can be declared to be within range Ø through 999, or within -99.99 through +99.99, but not to be within the range, say, -5 through 12.

c. Bringing COBOL into compliance with D4 would have major impact on both language and implementation. This is partially due to the fact that COBOL deals with data in a variety of formats, and providing range specifiers for each format would be a complex task. COBOL does not have a systematic exception-handling capability; thus, the user would have to define range exception handlers on a statement by statement basis in a similar fashion to the way in which ON SIZE ERROR is treated.

5. Variable Types (D5).

a. Degree of compliance: P

b. There are various restrictions on the way that data can be composed which prevent COBOL from complying with this requirement. For example: tables are restricted to three dimensions; tables must be embedded within records;

condition variables cannot be arrayed; flexible tables cannot be components of other tables.

c. While improvements with regard to this Tinman requirement are possible, full compliance would involve an entire re-orientation of COBOL (as indicated in connection with A1 above).

6. Pointer Variables (D6).

a. Degree of compliance: F

b. COBOL has no pointer facility nor any language-supported means for building data with shared or recursive substructure. In practice, COBOL users deal with some forms of linked data structures externally (e.g., on a disk). The applications for COBOL typically involve quantities of data considerably larger than can fit in memory. Thus, the approach generally taken in COBOL is to process each record individually, keeping only summary information between records.

c. Inclusion of a pointer facility in COBOL would be a substantial extension requiring large design effort. The achievement of data type security would present particular difficulties.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: F

b. COBOL lacks the facilities needed to allow user definition of new data types and operations. The only definitional features in COBOL are: (1) a facility to define structured data objects (i.e., records and tables), and (2) a limited ability to define callable subroutines. These are in no way sufficient to meet the requirements of E1; implicit in the notion of data type are restrictions on the uses of its member objects, but in COBOL there is complete freedom to use composite data-items simply as alphanumeric values regardless of their structure.

c. The absence of definitional facilities is one of the great weaknesses in COBOL. There is no way to bring the language into compliance with E1 without a nearly complete redesign.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. Lacking user-defined types, COBOL fails to meet this requirement.

c. The addition of user-defined types of the kind implied in E1, E5, E6 and E8 is a major change requiring a reorientation of COBOL's data handling facilities. The problem with COBOL here is the language's focusing on the character string as the atomic constituent of (almost) all data structures.

3. No Default Declarations (E3).

a. Degree of compliance: T

b. COBOL satisifies this requirement fully.

4. Can Extend Existing Operations (E4).

a. Degree of compliance: F

b. COBOL has no provision for extending existing operators.

c. Operator extension implies the existence of a type definition facility; the latter is absent from COBOL and would require major changes if it were to be included.

5. Type Definitions (E5).

a. Degree of compliance: F

b. COBOL has no facilities for defining new data types.

c. The provision of a type definition facility for COBOL is a major change and would require redesign of a substantial portion of the language.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: PF

b. COBOL partially fulfills this requirement (with respect to structuring data, however, and not to defining new types). Definition by enumeration of literal names is present to a limited extent in the form of condition variables -- the literal names can only be used in conditions and may refer only to a relation between the enumerated value(s) and one particular data item. (It may be noted that, rather than defining an abstraction independent of its representation, the COBOL construct makes explicit the representation.) COBOL's record and table definition facilities capture, in a restricted fashion, the notion of Cartesian product (but only up to three dimensions for tables). There is no facility for discriminated union in COBOL; nor is there any provision for obtaining power sets.

c. The provision of these facilities in COBOL would be a major change to the language. Again, the problem is COBOL's basic orientation around character data.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The various facilities for overlaying data (see A7 above) effectively provide COBOL with free union. There is no type subsetting allowed in COBOL.

c. Restrictions on overlaying data would be a difficult change to make to COBOL, in view of the applications which require much overlaying for efficiency reasons.

8. Type Initialization (E8).

a. Degree of compliance: F

b. COBOL contains no facilities for meeting this requirement.

c. The provision of a type definition facility which satisfies E8 is a major modification to COBOL.

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1)

a. Degree of compliance: FP

b. COBOL is very weak in the area of namescopes; e.g., there is no block-structure, and all data names are known globally throughout the program. The only aspect of COBOL which approaches satisfying the intent of F1 is the "own"-like behavior of data in CALLed programs. As explained in [8, p. XII-5]: "A called program is in its initial state the first time it is called... On all other entries into the called program, the state of the program remains unchanged from its state when last exited. This includes all data fields...."

c. The addition of namescopes to COBOL would be a major change. The problem is that COBOL's PROCEDURE DIVISION is basically linear as opposed to hierarchical in structure.

2. Limiting Access Scope (F2).

a. Degree of compliance: F

b. COBOL contains no facilities for limiting access scope.

c. This Tinman requirement implies a data abstraction capability that does not exist in COBOL. Adding such a capability would amount to a substantial redesign of COBOL.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: FP

b. COBOL partially satisfies this requirement, since most names have their meanings identified at compile-time. The discrepancies between the language and F3 lie in two areas. First, the block structure implied by the requirement that "access scopes will be lexically embedded" is absent from COBOL. Second, some names are only known at run-time. As explained in [8, p. XII-1], COBOL "provides the capability to transfer control to one or more programs whose names are not known at compile time...."

c. Providing block structure in COBOL would be a major change, as mentioned in connection with F1.

4. Libraries Available (F4).

a. Degree of compliance: P

b. COBOL has a limited library feature which serves in effect as a simple macro facility for text insertion. Its main use appears to be a shorthand way of incorporating commonly used file or record descriptions into a program. It is not clear that this satisfies F4's requirement that there be "broad support for libraries common to users of well-recognized application areas."

c. The subroutine capability of COBOL is currently quite weak; thus, there is little advantage to be gained from an object library capability until the language more effectively supports procedures (a major change). On most of the large commercial machines which run COBOL there are general provisions for object time libraries.

5. Library Contents (F5).

a. Degree of compliance: PT

b. COBOL partially satisfies this requirement. The fact that accessing a library entails copying source text implies that the library contents are restricted to COBOL source language; however, the ENTER statement [8, p. II-63] allows code from any implementor-allowed language to be included in the source text.

c. There are no provisions in [8] for compile-time checks at the interface, and, in fact, the facility provided by ENTER is implementation-dependent.

d. The addition to COBOL of security checks on the interface between the language and ENTERed code is relatively minor with respect to language definition; impact on implementation is moderate.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: T

b. COBOL has no Compool; however, the library facility captures the essential power of the Compool and hence COBOL can be considered to satisfy F6.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. It provides interfaces to machine-dependent capabilities (e.g., file descriptions and the ENVIRONMENT SECTION), but does this via a large set of special cases as opposed to a few general rules.

c. The COBOL language could be modified only with difficulty to meet this Tinman requirement. Although COBOL attempts to encapsulate in the ENVIRONMENT DIVISION the distinctions between devices, many differences in file organization and processing must be reflected in the DATA and PROCEDURE DIVISIONs. COBOL tries to achieve program self-documentation through an English-like exposition. However, this has had the effect of emphasizing differences of organization in file and I/O descriptions rather than the features which are common to all I/O.

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. The sequential control structure is the normal flow of control. For conditional execution, the IF and GO TO ... DEPENDING statements can be used. The PERFORM statement handles iterative execution. There are no facilities for recursion or parallel processing. Several specialized features deal with exception and interrupt handling, primarily in connection with I/O.

c. The addition of recursion and parallel processing capabilities to COBOL would involve major definition and implementation difficulties. The strength of COBOL is that it takes a direct and simple approach to commercial calculations. It is straightforward in COBOL to process single records and to maintain summaries of information from the records processed. COBOL is not directed at real time applications nor at applications requiring sophisticated mathematical calculations; the inclusion of either of those capabilities would require a major effort and imply a different orientation to the language.

2. The GoTo (G2).

a. Degree of compliance: PT

b. The GO TO statement in COBOL satisfies this requirement [8, p. II-65]. It should be noted, however, that in the absence of namescopes COBOL permits GO TO statements to branch to any paragraph (or section) in the PROCEDURE DIVISION. Also, the ALTER statement [8, p. II-57] should not be used; it allows dynamic determination of the target of a GO TO, from a point which may be remote from the actual GO TO statement. This makes the dynamic behavior of a program quite difficult to perceive from its static structure; the GO TO ... DEPENDING statement should be used instead of the ALTER.

c. The ALTER could be easily eliminated as it provides no capability not provided by GOTO ... DEPENDING. The removal of the ALTER statement should simplify COBOL implementations.

3. Conditional Control (G3).

a. Degree of compliance: P

b. COBOL's conditional control structures satisfy this requirement to a high degree. The IF statement [8, p. II-66] is almost "fully partitioned" in the sense of the Tinman; the only exception is that the ELSE clause may be omitted if the IF statement is the last statement in the sentence. Selection may be based on the value of a Boolean expression, or on a computed choice, but not on the subtype of a value from a discriminated union (COBOL lacks discriminated union). A problem with the IF statement is that one cannot transfer control out of a single nested IF statement without transferring out of the entire sentence.

c. The GO TO ... DEPENDING statement [8, p. II-65] serves as a CASE statement. However, COBOL restricts the dispatch expression to be an identifier instead of a general formula. Also, the "fall-through" behavior in the presence of an out-of-bounds identifier is conducive to programmer error.

d. The modifications necessary to bring COBOL into compliance with G3 are not major (unless one interprets this characteristic as requiring a discriminated union facility). Generalizations of the IF and GO TO ... DEPENDING statements are needed.

4. Iterative Control (G4).

a. Degree of compliance: P

b. COBOL's PERFORM statement partially satisfies this requirement. In particular, it allows the repeated execution of one or more paragraphs or sections, either a fixed number of times based on the value of a literal or identifier, or until some condition becomes true. Several of the formats provided for the PERFORM statement are quite readable; e.g., PERFORM ... n TIMES (when a loop control variable is unnecessary), and the AFTER phrase (which nicely reveals the order in which loop variables are varied). However, the UNTIL phrase in the VARYING clause is not as clear as it could be; to execute paragraph P 10 times, with I as loop control variable, one must write: PERFORM P VARYING I FROM 1 BY 1 UNTIL I>10.

c. Some conflicts between COBOL and G4 are as follows. First, COBOL does not permit the loop termination condition to appear at arbitrary points in the loop. In fact, if one wishes to check for a terminal condition, he must code this in the following fashion:

```
LOOP. ... IF termination-condition THEN GO TO LOOP-EXIT. ...
LOOP-EXIT.  EXIT.
```

and then PERFORM LOOP THRU LOOP-EXIT to effect the iteration. A second conflict is that in COBOL all loop-control variables are global data. Third, it is possible to transfer control into the body of a loop from outside (interestingly, COBOL prohibits the transfer of control in the other direction).

d. In a COBOL program all identifiers are global to the entire program, because there is no way to restrict the scope in the PROCEDURE DIVISION of items declared in the DATA DIVISION. The loop itself is remote from its point of invocation and behaves more as an internally nested procedure than a loop but with the weakness that there is no 'IF done THEN return' possibility. Changing the loop construct would involve large changes to the COBOL PROCEDURE DIVISION.

5. Routines (G5).

a. Degree of compliance: F

b. Recursion is prohibited in COBOL; as stated in [8, p. XII-6], "a called program must not contain a CALL statement that directly or indirectly calls the calling program."

c. COBOL has two mechanisms for dealing with routines. If it is necessary to pass parameters, then the routine must be written as a separate COBOL program, invoked via a CALL ... USING statement. If parameter passing is not needed, then the routine may be written as simply a section or paragraph (called a "procedure" in COBOL) in the program's PROCEDURE DIVISION, invokable via a PERFORM statement.

d. All COBOL storage uses a static or own-like allocation scheme. Modifying the language to use the automatic storage allocation techniques and the displays necessary for recursive procedures would require a major redesign of the language. For typical COBOL applications, recursive procedures offer no particular advantages.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. COBOL contains no parallel processing facilities.

c. Parallel procesing implies that two or more separate procedures, which share at least some data, are simultaneously active. For a language to provide parallel processing facilities means that the shared data must be protected from overlapped update. For COBOL this might mean adding a "SHARED" section to the ENVIRONMENT and DATA DIVISIONS and mechanisms for starting parallel tasks. Such modifications would be a major extension to the language.

7. Exception Handling (G7).

a. Degree of compliance: P

b. COBOL provides a limited set of features for handling exceptional conditions of various kinds. The three basic types of conditions which can be handled are arithmetic overflow, I/O exceptions, and storage bounds overflow. To deal with arithmetic overflow (more precisely, a condition when an arithmetic result exceeds the capacity of the target) the user can supply an ON SIZE ERROR phrase [8, p. II-50] in any arithmetic statement. In the event of overflow, the imperative statement supplied in the ON SIZE ERROR phrase is executed and control then returns to the next statement.

c. A number of features deal with the handling of I/O exceptions. A two-character FILE STATUS data item is available for inspection after most I/O operations to determine the outcome (e.g., Successful Completion, At End, Permanent Error, or Implementor Defined, in the case of Sequential I/O [8, pp. IV-1-2]). The AT END phrase may be supplied to give a programmer-supplied behavior for conditions such as attempting to read when no next record exists [8, p. IV-28]. The USE statement "specifies procedures for input/output error handling that are in addition to the standard procedures provided by the input-output control system" [8, p. IV-28]. A procedure associated with a USE statement is executed after the standard system procedure for dealing with the exception. Procedures associated with USE statements are called

"declaratives" [8, p. I-99] and are logically and physically distinct from "non-declaratives."

d. Another possible exception condition in COBOL is storage overflow during the loading of a CALLed program. "If during the execution of a CALL statement, it is determined that the available portion of object time memory is incapable of accomodating the program specified in the CALL statement and the ON OVERFLOW phrase is specified [in the CALL statement], no action is taken and the imperative-statement [in the ON OVERFLOW phrase] is executed" [8, p. XII-5].

e. Because storage in COBOL is statically allocated, the only excessive memory requirements possible are during a CALL to a non-resident routine, which has an explicit error routine assigned. The arithmetic overflow condition requires the user to specify with each statement the exception handler; thus, there is no transfer of control either forward or backward (unless there is a GOTO). However, this explicit method of handling exceptions offers better opportunities for recovery than, for instance, PL/I, where one process must handle all occurrences of a class of exception conditions.

f. User definition of exception conditions would be difficult in COBOL because the procedure defining mechanism is so weak.

8. Synchronization and Real Time (G8).

a. Degree of compliance: F

b. COBOL was not designed to be a real-time language, and the inclusion of real-time facilities would be a difficult task.

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

a. Degree of compliance: P

b. COBOL partially complies with this requirement; areas of conflict are the following. First, COBOL is not free-format; the card-orientation of the language is evidenced by the rules in [8, pp. I-105 ff] (e.g., regarding the indication of continuation lines and the permitted placement of division and section headers). Second, there is an abundance of unique notation for special cases (e.g., the various statements associated with I/O); a simple, uniform grammar appears unachievable. Third, there are actually two types of statement separator: a semi-colon is optional between statements, and a period (followed by a space) is mandatory at the end of a sentence (a sentence comprises one or more statements). Fourth, COBOL allows abbreviations for keywords (e.g., CORR for CORRESPONDING, COMP for COMPUTATIONAL).

c. It might also be pointed out here that the lexical structure of COBOL [8, pp. I-75 ff.] is somewhat more complicated than it need be (especially regarding the placement of spaces). A finer subdivision of the lexical units -- e.g., including non-numeric literals as a token type, in addition to separators and character-strings -- might have been cleaner.

d. Many of the shortcomings that COBOL has in its lexical and syntactic structure are the direct result of the attempt to use English-like sentences wherever possible. Thus COBOL has a large number of noise words, has alternate constructs for singular and plural cases (e.g., RECORD IS and RECORDS ARE), requires a space after periods and provides other examples of clumsiness or verbosity.

e. The changes needed to bring COBOL into compliance with H1 are fairly major in scope.

2. No Syntax Extensions (H2).

a. Degree of compliance: T

b. COBOL fully satisfies this requirement.

3. Source Character Set (H3).

a. Degree of compliance: T

b. COBOL satisfies this requirement. Besides the 26 alphabetic and 10 numeric characters, COBOL includes the following:
space + - * / = $, ; . " ( ) < >
all of which are included in the 64-character ASCII subset. It should be noted, however, that COBOL allows in comments and non-numeric literals any character in the host computer's collating sequence.

4. Identifiers and Literals (H4).

a. Degree of compliance: P

b. COBOL satisfies this requirement almost completely. The formation rules for identifiers and literals [8, pp. I-75ff] are fairly conventional, except that data names may begin with a numeric character and that the break character for identifiers is the hyphen. (Because of the latter, spaces must appear around arithmetic operators, to distinguish A-B from A - B.) The only discrepancy between COBOL and H4 is that COBOL does not contain a break character for literals.

c. Providing a break character for numeric literals is not a major change; the problem is which character to use. For example, underscore is a possibility, but it can be confused with a hyphen and prints as a left arrow on some terminals.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. COBOL partially complies with this requirement. End-of-line is allowable in non-numeric literals [8, p. I-80]; continuation of lexical units across lines is also permitted [8, p. I-106].

c. The changes needed to have COBOL comply with H5 are minor.

6. Key Words (H6).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement; the primary conflict is that COBOL contains an extremely large number of reserved words (about 300, itemized in [8, pp. I-109-110]). As a result, the programmer must continuously be careful when devising data-names, to avoid conflicts with this set. The opposite problem is also present; the language has in some cases sacrificed clarity by avoiding the use of reserved words. In particular, levels 66, 77, and 88 for data-names have special meanings which are not at all suggested by the numeric values.

c. COBOL programs are intended to be readable by people whose main areas of expertise lie outside the field of computer programming. This goal has led to an inappropriate reliance on English-like constructs (see the discussion of H1 above) and an excessive number of key words, all of which are reserved.

d. It would be very difficult to bring COBOL into compliance with H6 without sacrificing some of the language's basic design goals. The impact on implementation, however, is minor.

7. Comment Conventions (H7).

a. Degree of compliance: P

b. COBOL partially meets the requirements of H7 with the following qualifications. First, there are actually two comment forms in COBOL: the comment-entry used in the IDENTIFICATION DIVISION [8, p. II-2], and the comment line denoted by an asterisk in the continuation indicator area. Second, the contexts in which comments may appear are restricted by the above forms (i.e., one cannot place a comment adjacent to (on the same line as) the statement to which it applies).

c. It would not be difficult to define a comment form for COBOL which does not require an entire line. However, this should be done in conjunction with other redesign which removes the card orientation from the language.

8. Unmatched Parentheses (H8).

a. Degree of compliance: T

b. COBOL complies with this requirement completely. Except when they occur in non-numeric literals and pseudo-text, parentheses "may appear only in balanced pairs of left and right parentheses delimiting subscripts, indices, arithmetic expressions, or conditions" [8, p. I-75].

9. Uniform Referent Notation (H9).

a. Degree of compliance: F

b. COBOL fails to satisfy this requirement. A standard subscript form is present in the language, but a different notation is used for calling routines (programs), viz., CALL program-name USING data-name-1, ... data-name-n. Moreover, there is no facility for calling a routine which returns a value (i.e., a function). In addition, COBOL distinguishes between subscripts and indexes: a subscript is any variable containing an integer, while an index (USAGE IS INDEX) and table (INDEXED BY) must be explicitly declared as such [8, p. I-89]. The difference between subscripts and indices is one of efficiency.

c. Major changes would be required to bring COBOL into compliance with H9; e.g., a "function" subroutine facility would be needed.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. One exception is the difference of interpretation applying to adjacent level-01 entries, depending on whether they appear in the FILE SECTION or the WORKING-STORAGE SECTION. (In the former case, storage overlaying is implied; in the latter case, separate storage is allocated.) A second exception is the two uses for the equal sign: equality in relations, and assignment in the COMPUTE statement. As another exception, the hyphen acts as the break character in identifiers, the subtraction operator (a potential source of confusion), and the continuation character for multi-line sentences, entries, phrases or clauses.

c. Different symbols for equality and assignment operators could be used (e.g., '=' and ':='). The hyphen could be reserved for subtraction, underscore used for the break character and one of the 'spare' characters used as a continuation character. The different interpretations of level-01 entries in the FILE SECTION and the WORKING-STORAGE SECTION could be most easily resolved by requiring a REDEFINEs-clause in the FILE SECTION when specifying alternate record formats.

Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: P

b. COBOL has numerous examples of implementation dependencies and "undefined" situations, in conflict with I1. The action to be taken by a compiler when a syntax error occurs is undefined in [8]. When no SIZE ERROR phrase is specified, the result of an overflow is undefined [8, p. II-50]. When the contents of a referenced data item are not compatible with the class declared for that item in its PICTURE clause, the result of the reference is undefined [8, p. II-52].

d. Full compliance would require that for every undefined action or result the standard specify an explicit action or result. This would be a difficult definitional problem. For some existing compilers the changes required could be very difficult because most of the actions to be defined involve exceptions or errors.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. COBOL partially fulfills this requirement, with respect to data representation. The standard (i.e., "default") representation of record components is contiguous character positions irrespective of word boundaries; also, the standard representation of numeric data is DISPLAY (i.e., character code) format. The programmer can override the first default via the SYNCHRONIZED or JUSTIFIED clauses, and the second default via the USAGE IS COMPUTATIONAL clause.

c. COBOL fails to provide any facility for the programmer specifying open vs. closed subroutines. For such a facility to be relevant, a more general subroutine definition mechanism is needed; this is a major change to the language.

d. With no parallel processing or real time capability, COBOL has no need for a reentrant vs. nonreentrant code

generation option. Addition of parallel processing or real time features would be a large extension to the language.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. COBOL fails to satisfy this requirement. Although the ENVIRONMENT DIVISION includes, in the CONFIGURATION SECTION, paragraphs in which the programmer specifies SOURCE-COMPUTER and OBJECT-COMPUTER, this information cannot be interrogated in the program in the manner required by I3.

c. The figurative constants of COBOL could be extended easily to include configuration information; similarly, some information from the ENVIRONMENT DIVISION could be made available as pseudo-figurative constants. This requirement assumes a compile-time macro capability not available in COBOL (see paragraph I4).

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. COBOL contains no such facilities. The only features which approach conditional compilation are the Debug module (WITH DEBUGGING MODE serves as a compile-time switch [8, p. XI-3]) and the COPY statement with REPLACING clause [8, p. X-2]. The former has the following interpretation [8, p. XI-1]: "When the WITH DEBUGGING MODE clause is specified in a program, all debugging sections and all debugging lines are compiled [normally].... When the WITH DEBUGGING MODE clause is not specified, all debugging lines and all debugging sections are compiled as if comment lines."

c. The COPY statement with REPLACING clause offers a macro-like ability to insert source text in a program while replacing all occurrences of a given character-sequence by another character-sequence.

d. The features described in paragraph b are not sufficient to meet this requirement. They do, however, provide a base that can with moderate effort be extended to provide the conditional compilation facilities required by I4.

5. Simple Base Language (I5).

a. Degree of compliance: F

b. COBOL fails to satisfy the most important aspect of this requirement, viz., the possession of a simple base language. Because of the absence of any extension facilities, the entire COBOL language is in effect the base. The implication of this situation is that powerful and complex features must be built into the language (e.g., SORT, MERGE, STRING, UNSTRING) since there is no way to obtain the desired behavior using the language's definition facilities.

c. In addition, COBOL violates the requirement that each feature provide a "single unique capability". Frequently, COBOL offers several ways of doing the same thing: e.g., in the presence of the COMPUTE statement, the other arithmetic verbs are redundant; synonyms are built into the language (DIVIDE...INTO and DIVIDE...BY, OF and IN, EQUAL and =); optionally specified keywords abound; several different forms are available for declaring the attributes of data-names (the PICTURE clause and USAGE IS INDEX).

d. Although some simplification of COBOL is achievable with only a minor impact on the language (see H1 and H6 above), the ideal of a simple base language is not a realistic goal. A major redesign effort would be required to reshape COBOL to the form envisioned in I5.

6. Translator Restrictions (I6).

a. Degree of compliance: P

b. COBOL partially meets this requirement; specified in the language definition are the following restrictions: the maximum number of table dimensions is 3 [8, p. III-1]; the maximum number of levels in a record is 49 [8, p. I-84]; the maximum number of characters allowed in a PICTURE character-string is 30 [8, p. II-18]; the maximum number of digits in a numeric value is 18 [8, p. I-76]; the maximum size of a non-numeric literal is 120 characters [8, p. I-80]. No restrictions are placed on the number of nested parentheses levels in expressions or the number of identifiers or statements in programs [8, p. I-6].

c. Additional restrictions could easily be imposed on the language and on implementations. The problem is that of deciding what these restrictions should be.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. COBOL fully satisfies this requirement.

Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES.

1. Efficient Object Code (J1).

a. Degree of compliance: P

b. COBOL partially meets this requirement. On the positive side, there are several facilities by which the user has direct control over the efficiency of the object code. On the negative side, COBOL has some intrinsic inefficiencies resulting from some of its rules, and it also contains over-general features which have run-time costs even when the full generality is not employed.

c. A variety of features are available to the programmer to control the efficiency of the object program:

(1) The abbreviation of relational conditions almost announces an optimization to the compiler: A < B OR C OR D would obviously be compiled so as to preserve A in a register, whereas some extra effort by the compiler would be required to handle (A<B) OR (A<C) OR (A<D) in the same manner.

(2) The COMPUTE statement can result in more efficient execution than the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. For example,

```
COMPUTE Z = W * X + Y
```

can be compiled efficiently with no need for a temporary storage location to hold the intermediate result W*X. Using the MULTIPLY and ADD statements, however, would require an additional variable; e.g., MULTIPLY W BY X GIVING TEMP; ADD TEMP, Y GIVING Z.

(3) The use of INDEXes instead of subscripts can improve run-time efficiency.

(4) Specification of COMPUTATIONAL USAGE for data to be used arithmetically will improve run-time performance.

(5) The programmer has direct control of storage overlays via the facilities of the Segmentation module [8, Section IX]; he can also control the

allocation of storage to separately compiled programs via the CANCEL statement [8, p. XII-7].

d. Inefficiency of object code can be caused by COBOL's rules concerning order of evaluation. For example, in the expression (A+B)/C**2, COBOL requires the parenthesized subexpression (A+B) to be evaluated first; the generated code might look like:

```
LOAD A
ADD B
STORE TEMP1
LOAD C
MULT C
STORE TEMP2
LOAD TEMP1
DIV TEMP2
```

If the compiler were free to evaluate C**2 first, there would be no need for the second temporary:

```
LOAD C
MULT C
STORE TEMP1
LOAD A
ADD B
DIV TEMP1
```

e. COBOL also contains facilities whose generality must be "paid for" even when not fully used. Examples are the INSPECT [8, p. II-68ff], STRING [8, p. II-86ff], and UNSTRING [8, pp. II-91ff] statements. The problem is that lacking definition facilities, COBOL must have a large number of features built in which could be obtained by extension in other languages. But to make these features useful to a wide range of programming applications, COBOL defined them in a highly general fashion, with the result that run-time overhead may ensue even when the full generality is not used.

f. Improving the efficiency of COBOL to bring it into compliance with J1 would involve a moderate amount of language redesign.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: TU

b. This requirement actually applies to the translator rather than to the language itself; COBOL has no conflicts with J2.

3. Machine Language Insertions (J3).

a. Degree of compliance: P

b. COBOL contains a facility for dropping into machine language (or other high-order languages) -- viz., the implementation-dependent ENTER statement [8, p. II-63]. The restriction that usage of this facility be limited to compile-time conditional statements is not met by COBOL.

c. If the compile time facilities required by I4 were available, this requirement would be satisfied. Inclusion of such facilities in COBOL would be moderately difficult.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. COBOL partially satisfies this requirement. Examples of compliance are the SYNCHRONIZED clause (allowing alignment of elementary items on word boundaries, either left or right); the JUSTIFIED clause (permitting programmer override of the normal conventions for positioning whole data-items in receptacles); the USAGE COMPUTATIONAL clause; and the ability to specify "don't care" character positions inside a record as FILLER [8, p. II-15].

c. There are, however, several aspects of COBOL which are in conflict with requirement J3. First, there is no separation of a record's logical description from its object representation; both are defined in terms of character positions within the record. Second, there is no provision for associating a source language identifier with special machine addresses. Third, there are no restrictions on the use of the machine-dependent characteristics of the object representation.

d. The changes needed to meet this requirement would be large, because of the language's orientation around character data.

5. Open and Closed Routine Calls (J5).

a. Degree of compliance: F

b. COBOL fails to provide any of the requested features.

c. In the absence of a flexible procedure facility (see paragraph C7 above), this requirement is not relevant. Addition of such a facility would be a large task.

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: F

b. COBOL's I/O facilities, segmentation (i.e., program overlay) feature, and many of its high level capabilities (such as SORT, INSPECT, and COPY) require both a large run-time support system and an operating system.

c. Changes amounting to a redesign of the language would be needed to bring COBOL into compliance with K1.

2. Program Assembly (K2).

a. Degree of compliance: P

b. The COPY feature partially satisfies this requirement in that arbitrary portions of COBOL source text may be inserted in COBOL programs. A weakness of this facility is that the library code, even if fully debugged, can exhibit different behavior in different instances depending on the declared attributes of the data names.

c. Separately compiled programs (in COBOL or other source languages) may be linked to produce a "run unit" [8, p. XIV-40]. However, as noted above in connection with C6 and C7, the procedure capabilities of COBOL are weak, and there is no type checking on parameter passage.

d. A Compool-like library facility (i.e., partial compilation so that attributes are fixed) would solve some of the problems with the COPY feature; this would be a major addition to the language. Inclusion of strict type checking between separately compiled modules would also be a large change.

3. Software Development Tools (K3).

a. Degree of compliance: PU

b. The debug feature of COBOL (see paragraph I4 above) partially satisifies this requirement. The definition of linkers, loaders and other desired tools is outside the scope of [8].

4. Translator Options (K4).

a. Degree of compliance: U

b. The COBOL standard does not address the availability of options such as those required by K4. Their inclusion would not have a large impact on the language but could have major effect on implementations.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: F

b. COBOL offers none of the required facilities.

c. To add such facilities to the language would require major changes to the language design and to existing implementations. As the Tinman notes, there are many opinions as to the desirability, usefulness, and proper form for assertions, assumptions, axiomatic specifications, etc. Until there is widespread agreement on these issues, the inclusion of features such as those listed in K5 in a widely-used ANSI standard language such as COBOL would be ill-advised.

Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

a. Degree of compliance: F

b. Supersets are explicitly allowed in [8, p. I-6]:

> An implementation that includes, in addition to a specified level of each of the functional processing modules and of the Nucleus, elements or functions that either are not defined in the American National Standard COBOL specification or are defined in a given level of a standard module not otherwise included in the implementation, meets the requirements of this standard. This is true even though it may imply the extension of the list of reserved words by the implementor, and prevent proper compilation of some programs that meet the requirements of this standard. The implementor must specify any optional language (language not defined in a specified level but defined elsewhere in the standard) or extensions (language elements or functions not defined in this standard) that are included in the implementation.

c. Substitution of the prohibition required by the Tinman for the qualification quoted above would be a minor change to the language but could have a large impact on implementations.

2. No Subset Implementations (L2).

a. Degree of compliance: F

b. COBOL totally fails this requirement. The language, as specified in [8], comprises twelve modules all of which have a level-1 and a level-2 implementation and some of which have a level-0 (meaning the module need not exist at all). As specified in [8, p. I-4]:

> The full American National Standard COBOL is composed of the highest level of the Nucleus and of each of the functional processing modules. A subset of American National Standard COBOL is any

combination of levels of the Nucleus and of each of the functional processing modules other than the full American National Standard COBOL.

c. Since COBOL was designed to be a subset-oriented language, compliance with C2 would be a major redesign effort.

3. Low-Cost Translation (L3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard. However, the baroque structure of COBOL and the different syntax conventions used in the different divisions make it unlikely that a COBOL compiler will achieve the goals of requirement L3.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in [8]. However, the history of COBOL usage reveals that the language has been implemented for a variety of object machines.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. The COBOL standard makes no mention of the action to be taken by the compiler in the event of syntax errors or violations of other language restrictions.

7. Diagnostic Messages (L7).

a. Degree of compliance: U

b. The COBOL standard makes no mention of error or diagnostic messages and (as noted under L6 above) there is no requirement that errors be detected. The translator's behavior is implementation-dependent.

8. Translator Internal Structure (L8).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

   a. Degree of compliance: P

   b. As an existing and widely used language, COBOL is composed of features that do not exceed the state of the art. In view of the wide discrepancies between COBOL and the Tinman characteristics, it is not reasonable to expect meeting the following Tinman requirement: "Any design or redesign [to COBOL] which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project."

2. Unambiguous Definition (M2).

   a. Degree of compliance: P

   b. Despite the attempts in [8] to provide a complete and unambiguous specification of COBOL, the complexity of the language makes such a goal virtually impossible to achieve. The following brief set of examples illustrates some of the problems in the document.

   (1) The semantics of INDEXes is confusing. If one declares a table to be indexed by a set of data names, the behavior is unspecified when a data name outside this set is used as an index for the table. Also, as noted above in connection with A7, [8] does not explain the effect of REDEFINEing a field which includes an INDEX.

   (2) Nowhere in the language definition is there an explicit requirement that all data names must be declared.

   (3) "Undefined" situations are frequent (e.g., [8, p. III-6], [8, p. II-50]).

   (4) As described in connection with C1 and C2, the language definition confused the independent concepts of the parsing of an expression into constituent subexpressions, and the order in which these subexpressions are evaluated.

(5) As observed in connection with C6, the description of parameter passing is incomplete.

3. Language Documentation Required (M3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard. However, because COBOL is the most widely-used HOL, introductory and tutorial literature is available. We do not know of any formal definition of COBOL; such a description would be extremely difficult to produce.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the COBOL standard.

Section XV. CONCLUSIONS REGARDING COBOL

1. Conflicts between Objectives of COBOL and Tinman.

a. Among the six candidate HOLs considered in this report, COBOL is the furthest from satisfying the Tinman requirements. This is not surprising, considering the vast difference between the goals of the two. COBOL was designed as a special-purpose business-oriented language; power was not considered a critical need. The sizable investment in COBOL programs has created a natural conservatism with respect to language modifications. Thus, there is no impetus to make incompatible changes to the language, despite the advances in programming methodology which have revealed better (i.e., more secure and more powerful) constructs than those supplied in COBOL. A result of this inertial effect is that changes to COBOL have tended to be of the "add-on" variety, and interactions with other features have led to a large and complex language.

b. The Tinman's "needed characteristics," on the other hand, are oriented to general purpose programming and attempt to incorporate the current state of the art in the production of reliable software. Obviously, there is no commitment in the Tinman to the constraint of compatibility with existing languages.

2. Summary of Major Areas of Conflict between COBOL and Tinman.

a. Data and Types. The absence of a Boolean data and the fact that the existence and treatment of floating point are implementation-dependent are serious deficiencies in COBOL. The clumsiness of the array facility is also a major drawback. Even in COBOL's selected area of business data processing, the lack of any secure method of processing records with alternate structures is a serious handicap.

b. Operations. The presence of implicit conversions and the absence of explicit conversions, and the lack of operations for dealing with power sets or bit strings, resulted in serious conflicts with the Tinman. COBOL also offers little in the area of scalar operations.

c. Expressions and Parameters. The main failure is with respect to C6 (Type Agreement in Parameters); COBOL lacks

type checking on parameter passage. Also, COBOL does not generally allow expressions in contexts in which constants and variable references appear (C3, Expressions Permitted), nor is there a facility for pre-run-time evaluation of constant expressions (C4, Constant Expressions). The discrepancies with respect to C8 (Formal Parameter Specifications) and C9 (Variable Numbers of Parameters) are less critical.

d. Variables, Literals, and Constants. COBOL does not allow the association of constant values with user-supplied identifiers, as required in D1 (Constant Value Identifiers). Apparently, such a facility did appear in previous versions of the language (the CONSTANT SECTION of the DATA DIVISION) but is not present in X3.23-1974. Also, COBOL lacks a pointer facility (D6, Pointer Variables).

e. Definition Facilities. The shortcomings of COBOL are most apparent in this area; COBOL does not permit the definition of new data types. Thus the language fails to comply with E1 (User Definitions Possible), E2 (Consistent Use of Types), E5 (Type Definitions), or E8 (Type Initialization). It also lacks the ability to extend existing operators (E4).

f. Scopes and Libraries. The absence of block structure is a serious deficiency in COBOL, especially with respect to the construction of large programs. This absence is reflected in COBOL's failure to meet requirement F2 (Limiting Access Scope).

g. Control Structures. COBOL's deficiencies in the area of real-time applications are revealed here; the language lacks facilities for parallel processing (G6) and for synchronization (G8). Also, there is no support in COBOL for recursive routines (G5).

h. Syntax and Comment Conventions. COBOL lacks a notation for uniform reference (H9). COBOL is not free format and has many special forms.

i. Defaults, Conditional Compilation and Language Restrictions. COBOL's deficiencies in this area are a lack of compile-time variables (I3) and conditional compilation (I4), and its failure to have a simple base language (I5).

j. <u>Efficient Object Representations and Machine Dependencies</u>. COBOL's deficiency here is relatively minor; it lacks a facility whereby the user can distinguish between open and closed subroutine calls (J5).

k. <u>Program Environment</u>. The major conflict with the Tinman in this area is COBOL's requirement for an operating system and large run-time support (violating K1). Also, COBOL lacks a variety of facilities, in connection with software development tools, translator options, and assertion-like specifications, which are required by the Tinman.

l. <u>Translators</u>. Major conflicts here arise because COBOL is designed to have subsets and supersets (in violation of L1 and L2), the complexity of the language makes low-cost translation an unrealistic goal (in violation of L3), and the reference document does not require translator checking or standardize a set of error messages (violating L6 and L7).

m. <u>Language Definition, Standards, and Control</u>. These requirements do not actually pertain to the language; we note, however, that the COBOL defining document has serious conflicts with M2 ("Unambiguous Definition").

3. Unnecessary Features in COBOL.

a. There are a large number of COBOL features unnecessary for meeting Tinman requirements and many other features which are realized in the language via redundant forms. Examples of the former are the INSPECT statement, the Sort-Merge Module, editing PICTUREs, the SEARCH statement, and the ALTER statement. Of these, we specifically recommend deletion of the ALTER statement in view of the difficulties this statement raises with respect to program verification and maintenance, and because the main effect of the ALTER statement can be achieved via the GO TO ... DEPENDING construct. As for the other features named, they are present in COBOL primarily because of the weakness of the language's definitional facilities. If they were to be eliminated, a considerable sacrifice in notational convenience would have to be made. As a result, we do not recommend their deletion. In view of the fact that some of these facilities are "special purpose" in nature, we recommend that COBOL's subset-oriented approach be preserved, despite the Tinman's opposition (L2).

b. The redundant forms in COBOL are illustrated by separate constructs which provide the same effect (e.g., the arithmetic and COMPUTE statements, or the MOVE statement and some options of the arithmetic statements), by "noise" words, and by abbreviations and spelling variations for the same keyword. Although the attempt here was to promote readability, the opposite effect can unfortunately result, since the person who maintains a program must be familiar with all the variations employed. We recommend the removal of redundant forms, and suggest that, until this is done via the language definition, COBOL project managers select a single variation for each form to be employed by the programmers.

4. Problems Concerning Verification of COBOL Programs.

A recent study [10] sponsored by the Army's Computer Systems Command treated the feasibility of formally proving the correctness of COBOL programs. Aspects of the COBOL language which were specifically cited for their difficulty with respect to verification are the ALTER statement [10, p. 13]; the Debug, Sort-Merge, and Communication Modules [10, p. 14]; storage overlaying [10, p. 18]; truncation [10, p. 26]; consideration of data items as strings; and implementation-defined language features [10, p. 28]. A conclusion of [10] was that the major problems encountered concerning COBOL verification were "verbosity of the programs, assertions, and verification conditions" and "the semantic complexity of the COBOL language". This conclusion is consistent with our evaluation of COBOL in this chapter.

5. Recommendations concerning COBOL.

On the basis of the evaluation conducted in this chapter, we conclude that an attempt to modify COBOL to bring it into compliance with the Tinman would be inadvisable. The many fundamental differences between the two, caused by the basically different objectives desired, would imply a substantial redesign effort and a new implementation. It is not realistic to expect the "flavor" of COBOL to be retained in such a new language. Thus, the advantages typically cited for choosing an existing language (availability of implementations, documentation, trained programmers) would effectively be forfeited. In summary, COBOL in its present form is not suitable with respect to satisfying the Tinman requirements, and no language which differs from COBOL in only minor ways will be substantially better.

CHAPTER 8

PL/I EVALUATION

Section I. LANGUAGE SUMMARY

1. Lexical Properties.

PL/I is a free-format language based upon the EBCDIC character set. A fifty-six character subset is used for token formation [2, Section 2.5]; however, in special contexts such as character literals and comments any character defined in the implementation may be used. The space character separates tokens. PL/I's 188 keywords are not reserved. Identifiers may be of arbitrary length, but an implementation may restrict the maximum length provided that the maximum is at least 31 characters. The underscore may be used as a break character in identifiers. PL/I comments are delimited by "/*" and "*/", and any implementation-defined character may appear in a comment. Comments cannot be nested; so, although code may be included in a comment, the included code may contain no comments.

2. Data Types.

The data types available in PL/I can be roughly categorized as shown in Figure 7.

a. Scalars. The two scalar categories are computational and non-computational. PL/I offers type-checking only between the computational and non-computaational types and among the non-computational types. For example, it is illegal to assign a label value to a floating point variable or a pointer variable.

(1) Behavioral properties of all non-computational data types.

(a) Assignment is defined for POINTER and OFFSET, ENTRY, LABEL, FORMAT, AREA, and FILE variables. On assignment PL/I will convert between POINTER and OFFSET, but in all other assignments involving non-computational types, the source and the target have the same type. (In effect, POINTER and OFFSET may be considered the same type.)

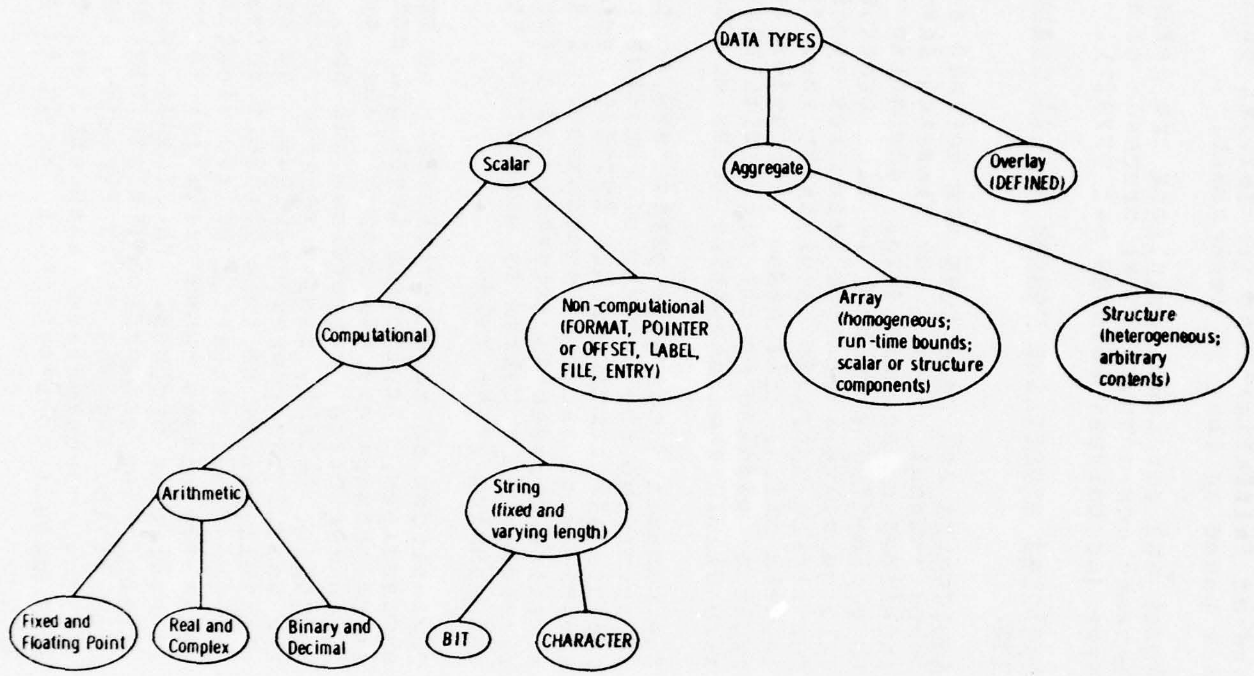

Figure 7. Data Types in PL/I

(b) The semantics of value parameters to procedures, of returning the results of function procedures, and of initializations (in INITIAL declarations) are based on that of assignment.

(c) Equal and not-equal relations are defined between non-computational operands of the same type (or between POINTER and OFFSET).

(2) Behavioral properties common to all computational types.

(a) Assignment and relations are defined among all computational types. Any mismatch involving an undefined conversion (e.g., assigning '12A.34' to a floating variable) or high-order truncation will generate a runtime exception condition. For string operands, assignment when fixed lengths differ will cause truncation on the right or padding on the right with blanks or zero bits. The conversion rules are complex.

(b) The semantics of value parameters to procedures, of returning the results of function procedures, and of initialization (in INITIAL declarations), are based on those of assignment. A type mismatch between an argument and a formal parameter is resolved by converting the argument and passing it by value.

(c) PL/I places no type restrictions on operands in expressions. Implicitly-invokable conversions exist between any two computational types. These are "true" conversions and not, as in JOVIAL or TACPOL, simply re-interpretations of the underlying representations. We emphasize the full extent of this implicit conversion process: concatentation of two floating variables causes those variables to be converted to character strings. The result is a character string. The expression (INT||FLOAT)**INT will convert the INT and FLOAT types to character strings, concatentate those strings, convert that result to float and raise it to the power INT.

(d) There are a number of built-in conversion routines, which give the PL/I user the ability to explicitly control precision, length and other attributes of the result.

b. Aggregate Data Types. PL/I has extremely flexible facilities for defining homogeneous or heterogenous data structures. Arrays may have any number of dimensions, and components may be of any scalar data type or any structure. Structures may have any scalar, array or structure components.

(1) Properties common to all aggregate data type.

(a) Component Selection occurs via indexing (for arrays) and via dot-qualified name (for structures). Only sufficient qualification to assure uniqueness is required.

(b) Compatible aggregate data types may appear in relational, string or arithmetic expressions; the operations are performed on a component-by-component basis. There are some restrictions: neither operand may contain a component of a non-computational data type and the two aggregate types must be compatible. See comments under paragraph J1 for more description. Note that "compatible" does not mean the two aggregate data types "look alike" (e.g., any computational scalar is compatible with all structures and arrays having no components of non-computational type).

(c) Assignment is defined for aggregate data types if the source is promotable to the target.

(d) Alignment is a representational property of aggregate types: the user may declare an aggregate UNALIGNED to minimize storage required or ALIGNED to minimize execution time.

(2) Arrays. PL/I permits the declaration of arrays of any rank of any component type (including structure). Bounds may be specified either at compile-time or at run-time. An array slice is permitted wherever an array of the slice's rank is permitted.

(3) Structures. A structure is PL/I's version of what the Tinman calls a record. PL/I permits the declaration of structures with components of any scalar type (computational or non-computational), array type or structure type. Each component has a name. Component selection is through dot-qualified and indexed identifiers (e.g., A(3).B.C(5).D, which could also be written A(3,5).B.C.D or A.B.C.D(3,5) as well as in other ways). Qualification sufficient to make the name unique is required; i.e., in the example above if D appears only once in the entire namescope, then D(3,5) selects the same component as A.B.C.D(3,5).

c. Untyped Data. PL/I contains several facilities which defeat the type checking for scalar data.

(1) The UNSPEC may appear as the target of an assignment.

(2) The DEFINED declaration (PL/I's version of EQUIVALENCE or OVERLAY) has the effect of a free union; it permits instances of scalar or aggregate types to be overlayed on the same storage area.

3. Procedures.

a. Function Procedures and Procedures. PL/I allows the definition of procedures. Procedures that have a RETURNS attribute are "function procedures" and may be used everywhere a reference is required. Procedures having no RETURNS attribute may be used only in CALL statements.

b. Formal Parameters. A procedure or function procedure may have any number of parameters. Each parameter may be of any scalar (computational or non-computational) or aggregate type. Function procedures may return values of any scalar or aggregate type.

c. Arguments. Non-computational arguments must match the type of the corresponding parameter (correspondence is by position in the argument or parameter list). The types of computational scalars or aggregates need not match. If they do not match, implicit conversions are performed as discussed under assignment. Converted arguments are passed by value; if the called procedure returns some result in one

of its parameters, then a type mismatch will cause these results to be lost with no exception condition raised. The result of an expression in the argument list is clearly passed by value; through the use of parentheses the user converts any reference into an expression and thus forces the argument to be passed by value. When attributes of parameters and arguments match exactly, the argument is passed by reference.

4. Exception Condition Handling.

a. PL/I provides a mechanism allowing the user to obtain control upon the occurrence of any of a set of language-defined, implementation-defined or user-defined exception conditions. The implementation-defined conditions relate to certain I/O and operating system actions that are dependent on configuration and hardware. The user gets control of an exception by executing, before occurrence of the exception, an ON-statment (e.g., ON FIXEDOVERFLOW: {<single statement>|<begin block>}). If, and whenever, a FIXED OVERFLOW occurs, then the <statement> or <begin block> is executed; it is executed only for exceptions and not executed sequentially. Note that control is transferred backwards, logically if not in the actual source text, when an exception occurs. After execution of the on-unit, control returns to a point just before or just after the code causing the exception unless there is an explicit transfer of control (a GOTO) from the ON-unit, in which case the ON-unit and any active blocks embedded in the block containing the ON-unit are terminated. Any condition may be raised through use of the "SIGNAL condition name;" statement. The user-defined conditions may be raised only through this means.

b. <u>Computational</u> <u>Conditions</u> are raised in the following:

(1) CONVERSION: The source for a conversion contains a character which cannot be converted to the target type (e.g., '12A.34' cannot be converted to any numeric type). The user can use ONSOURCE and ONCHAR to read and change the offending source string or character.

(2) OVERFLOW: Floating-point exponent exceeds implementation maximum.

(3) STRINGRANGE: Substring referenced with SUBSTR does not lie within the specified string.

(4) SUBSCRIPTRANGE: A subscript expression is not within declared array bounds.

(5) ZERODIVIDE: Fixed or float divide-by-zero error.

(6) FIXEDOVERFLOW: Truncation of high order bits or digits.

(7) SIZE: On formatted output or assignment, high-order digits are being lost.

(8) STRINGSIZE: On formatted output or assignment, a string value is too long to be represented in the specified field or variable.

(9) UNDERFLOW: A floating point computation yields a value not equal to zero but which is too small to be represented in the object machine.

c. I/O Conditions. In PL/I an I/O condition is always associated with a particular named file. I/O conditions are raised in the following circumstances:

(1) ENDFILE: A READ or GET statement has read the end-of-file marker.

(2) ENDPAGE: For a PRINT file only, the user specified number of lines per page has been output.

(3) NAME: On executing a GET DATA statement the name in the input streams did not match any expected name.

(4) KEY: During a KEYed access operation, a key error has occurred.

(5) TRANSMIT: On any input or output operation there is a physical error (i.e., hardware error) which prevents completion of the requested transfer.

(6) UNDEFINEDFILE: During an open operation the requested file was not found or could not be created.

(7) RECORD: The program has attempted to transfer a record of a size unacceptable to the implementation or of a different size from the destination variable.

d. General Exception Conditions. These are conditions which can be raised for errors in processing that are neither computational nor I/O. These conditions are also raised on the occurrence of an exception condition for which the user has provided no explicit condition handler. General exception conditions are raised in the following circumstances:

(1) ERROR: Raised for any exception condition for which there is no explicit condition handler. ERROR is also raised when an explicit condition handler does not correctly process a condition.

(2) STORAGE: Raised when there is not adequate free storage available during a procedure prologue or an allocate statment.

(3) FINISH: Raised immediately prior to normal or abnormal program termination. In the absence of a FINISH condition handler, the program is terminated.

(4) AREA: Raised when an attempt to allocate a BASED data object within an AREA variable fails because there is insufficient remaining storage in the AREA.

e. Programmer-Defined Conditions. The user may declare a condition name, may use that condition name in an ON statement, and SIGNAL occurrence of the condition at any point in the program.

5. Statements.

a. Null Statement. This statement permits extra semicolons to appear in programs.

b. Assign Statement. The semantics of assignment was described above; the PL/I assignment statement permits multiple target variables which are assigned to in left-to-right order (e.g., I=4; A(I),I,A(I) = 5; sets A(4), I, and A(5) to 5).

c. Call Statement. This statement is used to invoke procedures which have no RETURNS attribute.

d. GOTO Statement. The PL/I GOTO statement may be used to transfer control to any label whose name is known in the scope of the statement, or, through a label parameter or label variable, to any label in an active block. Transfers out of procedures are possible but jumps into nested blocks are not. Arrays of label constants are possible and may be used for an effect similar to the FORTRAN computed GOTO.

e. IF Statement. The PL/I IF-statement may have either one alternative (following THEN) or two alternatives (one following THEN and the other following ELSE). The expression following the IF is eventually evaluated to a bit-string and a high-order 1-bit is the "true" condition.

f. DO Statement. The PL/I DO statement may serve two different purposes. It may be used simply as parentheses for a group of statements which is to be treated as a single statement but without the overhead of entering a nested begin block (e.g., a group of statements to be executed for the THEN clause of an IF-statement). The other purpose is to allow iterative execution of a sequence of statements. It can do this in a variety of ways: by executing while some Boolean expression is true; by stepping a loop control variable from an initial value by fixed amounts until a terminal value is reached; by evaluating an expression after every iteration and assigning the result to a loop variable or by a combination of the three. The loop control variable may be of any computational type.

g. Begin Blocks. A sequence of declarations and statements may be bracketed by BEGIN and END to form a begin block, which may then be used anywhere a single statement is valid. A begin block delimits a namescope.

6. Storage Allocation.

PL/I has four storage classes, each of which implies a different allocation strategy. STATIC storage is allocated before execution. AUTOMATIC storage is allocated as part of the prologue to a begin block or procedure. AUTOMATIC storage is de-allocated, FREEd, on normal or abnormal exit from a begin block or procedure. BASED and CONTROLLED storage are allocated by an explicit programmed ALLOCATE

statement and de-allocated by a FREE statement. Objects of all four storage classes are initialized at allocation time if the INITIAL attribute is coded as part of the declaration. STATIC strings and arrays must have their bounds fixed at compile-time, but strings and arrays of the other three storage classes may defer resolution of their bounds until allocation time.

7. Files and I/O.

a. One of the most attractive features of PL/I is the variety of features available for processing external data. The two major classifications of data files are STREAM and RECORD. STREAM files are character oriented; RECORD files are record (or block) oriented.

b. STREAM files are input, output or printed. Printing assigns some additional attributes such as page size, i.e., maximum number of lines on the page. STREAM files are processed using GET (read) or PUT (write) statements. The user may specify any of three different modes: EDIT, which performs formatted I/O; LIST, which performs I/O using default format specifiers; and DATA, which requires or supplies variable identifiers to perform I/O using default format specifiers.

c. RECORD files are collections of records, each of which contains a single scalar or aggregate data object stored in the machine's internal representation. There is no conversion during RECORD I/O. Record files also have one of the attributes DIRECT (i.e., random) or SEQUENTIAL, and may also have a KEYED attribute, which specifies that records are to have an associated key allowing the program to specify the next record to be processed.

d. In addition to the GET, PUT, READ, WRITE, and REWRITE (i.e., update) statements outlined above, PL/I has OPEN and CLOSE, DELETE (which deletes a record) and LOCATE (which allows the user to build an output record directly in the output buffer).

e. No data type information is preserved with a file, a fact which makes it possible to WRITE data of one type and READ it back as another. A file may be OPENed for input with attributes entirely different from those used in the creation of the file. Misuse of this behavior may cause an error at some point during file processing.

f. A number of exception conditions for I/O are defined. See paragraph I.4.c above.

8. Compile-Time Facilities.

a. PL/I has a %INCLUDE statement [2, Section 2.5.7] but no action for this statement is defined in [2]. In existing implementations, %INCLUDE inserts source text from a named file into the program; typically, this source text includes system-wide declarations and procedure definitions.

b. PL/I as defined by [2] provides no facilities for macros, conditional compilation, or COMPOOLS.

c. Although [2] defines the translation and processing with no reference to separate compilation, existing implementations may and do allow for separate compilation.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement in that an associated set of "attributes" fully specifies each PL/I variable and each component of a composite data structure; additionally, the attributes specify the precision, the storage allocation mechanism to be used (see F1 below), namescope, and, to a certain extent, the internal representation of a variable or component of a composite data structure. The attributes (and thus the type) of each variable are fully determinable at compile-time.

c. PL/I fails this Tinman requirement by:

(1) Allowing default declarations of attributes. Indeed, the attribute structure of PL/I is so complex that explicitly declaring all the attributes would be an onerous burden on the user.

(2) Providing implicit type conversions, which effectively remove the possibility of using type information for redundancy.

(3) Providing free union (the DEFINE) facility. There is no provision at run time for determining which alternative of a union is the active one. Certain uses of based storage effectively provide free union.

d. The following modifications would improve the conformance of PL/I to the Tinman requirement.

(1) Require explicit declaration of all type attributes (e.g., FLOAT, FIXED, BINARY, DECIMAL, PRECISION, etc.). The factoring allowed in the PL/I DECLARE statement removes part of the burden this modification might have for the user.

(2) Fix in the standard some storage class and namescope assumptions (e.g., AUTOMATIC, INTERNAL), which would then require explicit overrides. This approach also saves keywords.

(3) Remove all implicit conversion from the language. All conversions must then be explicitly specified using the built-in functions.

(4) Apply restrictions to the DEFINE facility (see A7 below).

e. The major impact upon implementation of these modifications is to simplify the construction of PL/I compilers, which would be somewhat smaller, faster, and more reliable. These modifications should also improve the quality of programs written in PL/I. Aside from d(4) above, these restrictions do not constrain the user in the operations that can be performed; only in how those operations are presented and how the work is divided between the compiler and the user.

f. It may be noted that the restrictions and modifications described above are contrary to some of the basic design goals of PL/I.

2. Data Types (A2).

a. Degree of compliance: PT

b. PL/I offers integers and floating values with implementation defined maximum precision, and fixed point real numbers with implementation defined maximum, precision and scale. PL/I provides bit and character strings which may be of fixed or variable length; bit and character strings have an implementation defined maximum length, and the variable length strings have a user-defined maximum length. PL/I offers the user the ability to define arrays and records (called STRUCTURES in PL/I), as required by the Tinman.

c. PL/I does not offer an explicit BOOLEAN type but uses a bit string of length 1 (see B6 for implications). Provision of an explicit Boolean type in PL/I would be straightforward.

3. Precision (A3).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement. A default precision may be specified to apply to all floating point variables in a namescope [2, Section 4.3.6]; such specification is not required. PL/I does not require precision specification for floating point variables; if not specified, precision defaults to some implementation-defined value [2, Section 4.3.6.3]. Precision applies to operations: an operand of an operator is converted to the precision of the operand with greatest precision [2, Section 9.3.2.1ff].

c. This Tinman requirement could be met by requiring that the default precision for a namescope be specified. A single specification at the outermost level would then suffice for an entire compilation. This is a relatively minor change.

4. Fixed Point Numbers (A4).

a. Degree of compliance: P

b. The fixed point numbers offered by PL/I have a range and a fractional step size which are specified by the user (or defaulted, as described above in connection with A1). This range is expressed as the number of bits or decimal digits used to represent the number. Thus FIXED DECIMAL (4,2) allows all values in the range -99.99 to +99.99. There is no way to restrict further the range (e.g., to between -50.00 and +50.00). The scale factors are managed by the compiler.

c. PL/I does not provide support for range restriction and testing. It is difficult to see how such a feature could be added to PL/I without great expense.

5. Character Data (A5).

a. Degree of compliance: P

b. PL/I offers no general facility for defining enumeration types. It is not possible, for instance, to use characters, as "characters," for index values. However, some of the effects of treating character sets as enumeration types are achieved: PL/I defines all the relational operators on single characters and on character strings, and PL/I does offer a number of built-in functions

allowing the programmer great flexibility in the processing of character data, including the following: translating between internal code and any specified code or between any two specified codes; finding the position of one string in another string [2, Section 9.4.4.35]; determining the collating sequence on the host machine [2, Section 9.4.4.14]; and selecting any specified substring of a given string [2, Section 9.4.4.69].

c. The modifications needed so that enumeration types can be defined are described in connection with paragraph E6 below. It should be pointed out, though, that an attempt to obtain the character data type by such a facility is not likely to be cleanly achieved. One problem is that a single character currently is obtainable as a character string of length 1, but there is no corresponding concept of a string of objects from an enumeration type. Associated with this problem is the issue of how a character string literal can be defined, since in general there is no literal form for arrays of enumeration types.

6. Arrays (A6).

a. Degree of compliance: P

b. PL/I requires explicit specification of the number of dimensions, and of the range of subscript values (unless the lower bound equals 1). The type of array component may be specified through default values [2, Section 4.3.6.3]. The number of dimensions and the type are determined at compile-time. PL/I allows the programmer to defer determination of both the upper and the lower subscript bound [2, Section 3.1.1 (A18, A129) and Section 7.2.7] until entry to array allocation scope. Only a contiguous subsequence of integers is allowed as subscript values [2, Section 7.2.7, Step 1.1]. An unspecified lower bound defaults to one -- not zero [2, Section 4.4.3.3, Step 2].

c. The modifications to PL/I required to meet this requirement either depend on other features (e.g., definition of an enumeration type) or are straightforward (e.g., fixing the lower subscript bound at compile-time or changing the default lower bound from one to zero).

7. Records (A7).

a. Degree of Compliance: F

b. Although PL/I permits records to have alternate structures, it fails to satisfy the Tinman requirement because no discrimination condition is required; thus the PL/I DEFINED attribute merely allows the same storage to be interpeted differently using different names. For instance, a PL/I programmer may declare a STRUCTURE, and DEFINE an alternate form and then use components from both the base STRUCTURE and the DEFINED alternate in the same expression. Type checking is defeated; the entire mechanism has very low security.

c. To fully satisfy this Tinman requirement, PL/I needs a modified declaration form to specify the alternate STRUCTUREs and the discrimination condition, and both compile-time and run-time enforcement mechanisms. These are major changes.

Section III. OPERATIONS

1. Assignment and Reference (B1).

a. Degree of compliance: P

b. PL/I defines assignment and reference operations for all data types. These operations are extended beyond the Tinman requirements in that automatic conversion (e.g., integer to float or character to numeric) and reformatting (termed "promote"ing in the PL/I standard) of structures [2, Section 7.5], are automatically performed (see also J1 below).

c. PL/I does not offer encapsulated type definitions; allowing encapsulated type definitions would require major changes to PL/I, as described in Section VI below.

2. Equivalence (B2).

a. Degree of compliance: P

b. PL/I will compare any two data objects (records, arrays, or atomic data) for identity. PL/I will convert -- implicitly -- all components of records or arrays before making the comparison [2, Sections 6.3.5.1 and 9.1.5]. Considering only the atomic data types, PL/I will compare any pair among floating point, fixed point, integer, picture, character string, and bit string -- performing whatever conversions are necessary.

c. The comparison of floating point values requires identity. The set of built-in functions could be easily supplemented to provide comparison functions having the required behavior.

3. Relationals (B3).

a. Degree of compliance: PT

b. PL/I defines the standard six relational operations [2, Section 4.4.4] for numeric data. These operators apply also to character and bit string data. PL/I does not have enumerated types. Unordered data (e.g., pointer or label values) may only be compared for equality [2, Section 9.3.2.5.1 Case 3].

c. The scope of modifications needed to add enumeration types to PL/I is described in connection with E6 below.

4. Arithmetic Operations (B4).

a. Degree of compliance: T

b. PL/I offers all the requested operations. When the programmer wishes to explictly control the precision of the result there are built-in functions -- ADD, MULTIPLY, DIVIDE, SUBTRACT -- which allow him to do this. When a programmer wishes to obtain the remainder from an integer division the MOD built-in function must be used.

5. Truncation and Rounding (B5).

a. Degree of compliance: F

b. Truncation is also considered in connection with B9 below. Range and precision are attributes of variables in PL/I, not of programs. In an assignment operation, the right-hand expression is evaluated before the attributes of the target are determined [2, Section 7.5.1]. Further, within an expression the evaluation is eventually between pairs of operands. In an infix-expression involving operands of different types a conversion is performed considering only the types of the two operands [2, Sections 9.1.5 and 9.3.2.1]. Thus, if A is a floating point variable, the statement

A = 13 + 1/4;

will assign 3.25E0 to A because the 1/4 is evaluated first to fixed point 0.2500...0 to implementation maximum precision with a scale factor of maximum precision minus one (i.e., (N, N-1) in PL/I-like notation) [2, Section 9.5.1.9, Step 3.2, and Section 9.3.2.4]. Then the addition will be performed and the result (precision, scale factor) are such that there is only one digit position before the decimal point [2, Section 9.3.2.1]. The most significant position is truncated <u>and</u> the "fixed-overflow-condition" is raised, if it is enabled. An argument could be made that the above operations are not within the range specifications of the program; however, fixed point division in PL/I is inherently so confusing that is is not possible to intuitively determine the result; hence the operation is error-prone.

c. There are several difficulties to be faced in changing the behavior of fixed-point arithmetic operations. One problem is defining suitable behavior [15, p. 87]:

> The precision rules are probably the part of PL/I that has received the greatest criticism. Much of the complaint has been supported by results that are apparently anomalous when considered outside the design philosophy of the rules. A great deal of effort has been expended in searching for a simpler and improved set of rules, particularly for division. Many different proposals have been examined, each has its own set of anomalous results and none has shown a sufficient improvement to justify an incompatible change.

Probably any well-defined behavior could be implemented; defining that behavior is the difficulty.

6. Boolean Operations (B6).

a. Degree of compliance: P

b. PL/I Boolean operations include only AND, OR, and NOT. A user can define the NOR operation in terms of these but this operation is not part of PL/I. A NOR operation on Boolean values or bit strings could easily be implemented.

c. AND and OR are not evaluated in short-circuit mode (*) [2, Section 6.3.5.1] because any expression is evaluated before (or independent of) any determination that the result will be assigned to a Boolean variable. Also, Boolean does not exist as a distinct type in PL/I but is represented by bit string of length 1. In longer bit strings only the high-order bit is examined [2, Section 6.3.5.1] to determine a Boolean result.

(*) The following is stated in [2, Section 1.2.1, Flexibilities of Interpretation 1.2.1.3.(4)]: "If an implementation can determine the result produced by evaluating an <expression> ... without evaluating it, the implementation need not perform the evaluation." Although this allows short-circuit mode, there is no requirement that Boolean expressions be evaluated in short-circuit mode. For reasons discussed in connection with B5, determination that short-circuit mode is appropriate is difficult under the language standard.

d. If Boolean is implemented as a separate data type then short-circuit mode could be easily implemented. Operations on bit strings cannot be short circuited; it would involve compiler overhead to determine that fixed-length bit strings were of length 1. For variable-length strings run time checking is required to find if short circuit mode is possible. To achieve full compliance with this Tinman requirement, the standard must require short-circuit mode rather than just allowing it. Without an explicit Boolean data type this would be a difficult modification; with Boolean, it is straightforward.

7. Scalar Operations (B7).

a. Degree of compliance: P

b. PL/I permits scalar operations and assignments on conformable arrays and data transfers between records or arrays of identical logical structure. PL/I also offers more: one-for-one compatibility of type is not required; records are not required to have identical form (see [2, Section 9.1.1.4] for definition of "compatibility" and [2, Section 7.5.3.1] for definition of "promotability") but rather be "compatible" or "promotable" depending on usage (see J1 below). Arrays involved in operations must have the same rank and each dimension must have the same number of components [2, Section 9.1.1.6, Step 2]. The components are not required to be of the same type.

c. PL/I multiplication of matrices is "unnatural, confusing and inconvenient" in exactly the manner used by the Tinman as an example [2, Section 9.1.1.6].

d. Transfers between arrays or records may be used to permit run time conversion from one object representation to another (e.g., to pack or unpack data).

e. The BY NAME option on an assignment statement allows the complete reformatting and conversion of data in a record.

f. Although PL/I allows obscure and confusing operations on arrays and records, great care must be taken in the design of restrictions (or redesign of the base features) because the highly complex behavior shown in the example in paragraph J1, for instance, results from the compounding of

much simpler behavior. If assignment of a scalar to a record is disallowed then much of the error-proneness also goes away. This is a sufficient restriction and would simplify implementation of new PL/I compilers. The changes to existing PL/I compilers would also be fairly minor, involving only restrictions in the definitions of compatibility and promotability applied.

8. Type Conversion (B8).

a. Degree of compliance: F

b. PL/I will attempt to convert -- implicitly -- between any two data types in the following list: float, fixed, integer, bit string, character string and pictured data (special form of character) [2, Sections 9.5ff]. The conversion from character to numeric will give an error if the character string contains any non-numeric character(s) [2, Section 9.5.1.7, Case 3].

c. PL/I provides explicit conversion operations: among integer, fixed and floating point data (FIXED and FLOAT built-in functions) and between object and character representations of numbers (GET and PUT STRING). The built-in function FIXED is also used for conversion between fixed point scale factors.

d. Restrictions on mixed mode expressions violate the spirit of PL/I but would not, in fact, reduce the power of the language or involve much extra writing in well-designed programs. The modifications needed to bring PL/I into compliance with B8 are relatively minor and should simplify the implementation.

9. Changes in Numeric Representation (B9).

a. Degree of compliance: PT

b. PL/I restricts ranges only through the precision and scale factor declared for fixed-point variables. Explicit conversion is not required between ranges so restricted.

c. PL/I provides a run-time exception (fixed overflow or size) when the most significant digits of any integer or fixed-point value are truncated [2, Sections 9.1.4 and 9.1.4.1]. However, as noted in paragraph B5 above, the

circumstances under which such truncation can occur are not intuitively obvious.

d. Although PL/I provides the desired behavior for this Tinman requirement, restricted ranges in the sense desired are not available; the addition of such range restrictions to the language would be a difficult modification to make cleanly.

10. I/O Operations (B10).

a. Degree of compliance: PT

b. One of the strong points of PL/I is the flexibility provided for I/O operations. PL/I files have either STREAM or RECORD attributes. Stream operations imply the processing (either reading or writing) of the next sequential character or compound operations processing groups of characters -- the data processed may only be character data. The PL/I programmer may treat character strings and stream files in a similar fashion. Record operations imply the processing of data in fixed or variable sized units (i.e., records). Records are transferred to/from the external medium without modification [2, Section 8.6.1.2]. PL/I provides an implementation-defined attribute, ENVIRONMENT, to describe any peculiarities of devices or file organization [2, Sections 1.2.1.6.(1) and 1.2.2.(5)].

c. Whether a PL/I program may dynamically assign and reassign I/O devices is an implementation-defined feature [2, Section 1.2.2(23) and (27)]. However, a file may be opened with certain attributes (e.g., OUTPUT STREAM), closed, and then reopened with different attributes (e.g., INPUT SEQUENTIAL RECORD).

d. The PL/I user may elect to process any specific I/O exception conditions and/or to process all exceptions in his program.

e. The only significant disagreement between B10 and PL/I is that PL/I allows a certain degree of implementation and installation dependence, which the Tinman prohibits. Complete implementation and installation independence would present some very difficult design problems.

11. Power Set Operations (B11).

a. Degree of compliance: P

b. PL/I does not offer enumeration types, so it does not offer power sets of those types. However, the PL/I bit string can be used by the programmer to represent user-defined power sets. The PL/I built-in function BOOL allows a user to define any bitwise operation between two bit strings.

c. Because bit string operations already exist in PL/I, implementation of power sets of enumeration types is greatly simplified. However, the definition of enumeration types and their behavior in PL/I would be quite complex.

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

a. Degree of compliance: F

b. The Tinman allows implementations to alter the order of argument evaluations when the alteration does not change the side effects of left-to-right evaluation. The PL/I standard explicitly allows certain potential side effects to be ignored. One potential side effect of an expression evaluation is the processing of exceptions (e.g., overflow or divide by zero). The standard does not require these exception conditions to come in any order [2, Section 1.2.1.4, items 1, 2, and 4]; moreover, implementations are not required to raise the exception conditions which are called for in the standard.

c. Functions are another source of potential side effects. If the function F(Y) alters X then the expression X**2+F(Y) could have different values depending on which of X**2 and F(Y) were evaluated first. [2, Section 1.2.1.4 item 3] states that the sub-expressions of an expression may be evaluated in any order.

d. The order of evaluation of expressions in PL/I could be fixed. That would make optimization more difficult. If the order of side effects (except those that arise from exception conditions) is fixed, then many useful optimizations still remain available. This would make new implementations easier to build and would not be difficult for existing implementations. Even compilers which perform substantial optimizations will usually disable the optimizations at the user's request.

2. Operand Structure (C2).

a. Degree of compliance: P

b. The PL/I arithmetic operator hierarchy is widely recognized and is similar to that provided by FORTRAN. The non-arithmetic operator herarchy is not so widely familiar but is similar to that provided by FORTRAN.

c. Because a string concatenation operation is not widely implemented, its position in the operator hierarchy

may not be widely recognized, but its PL/I assigned position between the arithmetic operators and the Boolean operators seems appropriate. PL/I has eight distinct levels of operator hierarchy. Operations within the same precedence level (e.g., X/Y/Z) will be parsed left-to-right (e.g., (X/Y)/Z) except at the highest level (e.g., X**Y**Z) where the operations are parsed right-to-left (e.g., X**(Y**Z)). This is standard FORTRAN and widely recognized; it also represents standard mathematical practice.

d. PL/I allows explicit parentheses either to specify the intended parsing or for clarity. PL/I never requires parentheses in an expression: regardless of how confusing an expression may be to a human reader, it is never ambiguous to the PL/I standard compiler.

e. PL/I offers no facilities for defining new operator precedence rules nor for changing the precedence of existing operators.

f. If the restrictions on mixed mode expressions discussed in connection with B8 were in effect, the potential complexity of a PL/I expression would be lessened. Another great improvement in clarity could be achieved by requiring the different classes of operators to be delimited by parentheses. For example, when the operands of a relational expression are arithmetic expressions, they must be parenthesized irrespective of the relative precedence between arithmetic and relational operators. These changes could be easily implemented.

3. Expressions Permitted (C3).

a. Degree of compliance: T

b. PL/I fully meets this requirement. A careful reading of [2, Section 2.4 and 2, Chapter 3] finds no instance of allowing references to variables of a particular type where expressions of that type are not also allowed. Constants are required, for example, to specify the precision of fixed point and floating point variables, the picture of picture variables, and the replication factor for a replicated string constant (e.g., the string 'AAAAAA' may be specified as (6)'A'), and in these positions variables are not allowed.

AD-A037 639 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2

CANDIDATE LANGUAGES EVALUATION REPORT.(U)

JAN 77 B M BROSGOL, R E HARTMAN, J R NESTOR DAHC26-76-C-0006

UNCLASSIFIED IR-217 USACSC-AT-76-11 NL

5 OF 6

ADA037639

4. Constant Expressions (C4).

a. Degree of compliance: P

b. The PL/I standard does not address this point explicitly. However, [2, Section 1.2] ("Relationships Between an Implementation and This Definition") taken together with [2, Section 1.2.1.4] ("Flexibilities of Interpretations Expression Evaluation") implies that an implementation has the option of evaluating constant expressions prior to run time. Although the PL/I standard could specify evaluation of constant expressions prior to run time, the constraints imposed in paragraph D2 might make such evaluation very difficult.

c. In places where PL/I requires a constant in the concrete syntax (e.g., [2, Section 2.4.4.1 CM27]), a constant expression would be a syntactic error. PL/I requires constants in few places; allowing constant expressions in those places would not be difficult. The problem is that of deciding on which operations should be compile-time evaluable in constant expressions (division? exponentiation? library routines?).

5. Consistent Parameter Rules (C5).

a. Degree of compliance: P

b. Parameters to PL/I procedures may be passed by value or by reference. All constants [2, Section 4.4.5.9 Create Entry Reference], expression results, substructured arrays [2, Section 4.4.5.10 Test Matching, Step 1], and all variables or aggregates requiring type conversion are passed by value. When the types (i.e., PL/I attributes) match, variables and arrays are passed by reference. When the forms of records and the types of all the corresponding components match, records are passed by reference.

c. An exception condition process (i.e., PL/I ON-unit) may be a single statement (not an if-statement [2, Section 2.3.3 CH9]) or a begin block, neither of which accepts parameters. There are built-in functions (ONCODE, ONCHAR, ONFIELD) which are only defined during an exception process. ONCODE returns an implementation defined value indicating the cause of the exception. ONCHAR and ONSOURCE change the contents of the character or the field causing the

conversion error. ONFIELD returns the entire contents of an input field when a data-directed I/O statement finds an unknown variable label [2, Section 8.7.1.5]. The use of these built-in functions as (essentially) arguments to on-units, and particularly their use as the target of an assignment statement, are inconsistent with the rules applicable for parameter passing to procedures. The non-I/O exception conditions have no parameters (except ONCODE) and can have no direct effect on the operation which generated the exception.

d. PL/I does not offer any parallel processing facilities.

e. The built-in functions of PL/I obey the same parameter passing rules as user-defined procedures, except that the MAX or MIN function can accept an argument list of arbitrary length and certain built-in functions may appear on the left side of an assignment statement (PL/I pseudovariables: IMAG, REAL, STRING, SUBSTR, UNSPEC, ONCHAR and ONSOURCE).

f. To bring PL/I into conformance with this Tinman requirement, the parameters to exception handling routines must be more carefully defined. To do so would involve some difficulty in both the definition phase and the implementation phase. The nature of the definitional problem can be seen in trying to determine where a parameter to an exception routine should come from, what it should do, and what should be done with it.

6. Type Agreement in Parameters (C6).

a. Degree of compliance: P

b. PL/I matches actual arguments and formal parameters by performing any conversions on the actual arguments necessary [2, Section 6.3.6.1.2]. The number of dimensions for array parameters is determined at compile time [2, Section 3.5.5]. In PL/I the size and subscript range for array parameters are passed as part of the arguments.

c. PL/I allows type transfers to be hidden in procedure calls [2, Section 6.3.6.1.2]. PL/I allows the subscript ranges for arrays to be implicit on the call side [2, Section 6.3.6.1.2 Step 2.3].

d. The rules against implicit type conversions discussed under paragraph B8 are sufficient to assure type agreement of actual arguments and formal parameters when all procedures are compiled together. The PL/I standard [2] assumes that all procedures are compiled together and does not discuss separate compilation and linking steps. Any changes to this larger environment could be very expensive. A linker in a large system is generally shared by many different compilers -- FORTRAN, COBOL, PL/I and Assembler -- so not only must the changes be made to the linker but possibly also to the other languages. Thus, the definitional phase presents some difficulties but the implementation could involve very large system changes.

7. Formal Parameter Kinds (C7).

a. Degree of compliance: P

b. PL/I offers the first formal parameter class (call by value) but does not disallow assignment to these parameters. Any change in the procedure to the parameter will have no effect on the argument, which may be a constant, an expression or a variable enclosed in parentheses. PL/I offers the second class of parameter (call by reference) but care must be taken by the programmer that the attributes of the argument and the parameter agree, or PL/I will perform a conversion and pass the converted value in a temporary location, thus creating the effect of a call by value.

c. It may be noted that control of argument class resides with the caller and not with the procedure. The caller can ensure call by value, which protects a variable from modification, but the procedure has no method of specifying that data to be modified is passed by reference.

d. There are formal parameter classes for labels, which may be used as exception condition exits, and for procedure names. User defined exception condition names remain valid within a namescope; thus, control in PL/I need not pass through the procedure call interface to process an exception condition.

e. Substantial changes to PL/I and to its supporting routines (e.g., linker) are required to allow procedures to specify which of their parameters will be modified and for the system to assure no violations to this intent.

8. Formal Parameter Specifications (C8).

a. Degree of compliance: P

b. PL/I offers an explicit generic facility in the definition of a procedure. Such a procedure must be internal [2, Section 4.3.6.1 Step 4]. The programmer defines a number of alternate entry points to a procedure; the translator selects the appropriate entry depending on the attributes of the arguments and uses that entry as the entry point to the named procedure [2, Section 4.4.5.11]. If the attributes of the call arguments do not match any of the sets of generic parameter attributes the compiler recognizes an error [2, Section 4.4.5.11 Step 1.2.5].

c. It is not clear that the capability required by the Tinman in fact offers the facilities desired. It is, however, quite clear that PL/I offers no "attribute-free" parameter class which would allow a procedure to build a stack or queue with arbitrary contents. A generic procedure with entry points for all conceivable argument types would be unwieldy.

d. It may be noted that a programmer could use PL/I's generic facility to suspend conversion, by specifying one entry with the calling argument attributes desired. Any deviations would result in an (implementation-dependent) error announcement.

e. A compile-time text substitution macro facility would best satisfy this requirement. Some versions of PL/I provide a macro facility, which was excluded from the standard because it is difficult to rigorously define. Provision of such a macro facility would not change other parts of PL/I but would be moderately difficult in itself.

9. Variable Numbers of Parameters (C9).

a. Degree of compliance: F

b. PL/I requires the number of arguments in the subroutine call or function reference to equal the number of formal parameters of the procedure [2, Section 4.4.5.9 Case 2.1]. There are some built-in functions having an arbitrary number of arguments (e.g., MIN and MAX) but this facility is not available for user-defined routines. PL/I does allow

different named entry points to the same procedures. When a different entry point is used, a different number of arguments may be supplied. With the generic feature, a single procedure name may be used with the selection of entry point dependent on number and characteristics of the arguments. In neither case is it possible in PL/I for a programmer to write a procedure which may be called with an arbitrary number of arguments.

c. Allowing an arbitrary number of arguments to a PL/I procedure, even with the restrictions imposed by the Tinman, requires a large change to PL/I.

Section V. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers (D1).

a. Degree of compliance: F

b. PL/I has no facility for representing computational constant value identifiers. Such a facility seems to be within the spirit of PL/I and should not be too difficult to implement, especially considering that PL/I does have a facility for processing constant value identifiers for non-computational values such as labels or file names.

2. Numeric Literals (D2).

a. Degree of compliance: P

b. PL/I provides a syntax and a consistent interpretation for literals of built-in data types; however, the relationship between the values of program literals and input data is not considered in [2].

c. The PL/I language standard could specify that numeric literals and input data of exactly the same form have the same value on any particular target machine. Such a change could have an effect throughout the entire compiler if host and object machines differ.

3. Initial Values of Variables (D3).

a. Degree of compliance: P

b. The PL/I initial attribute allows the programmer to specify the initial values of individual variables, arrays or records as part of their declaration [2, Section 3.1.5 A24, A25, A26]. These variables are initialized at the time of their apparent allocation [2, Section 5.3.6 Step 1.1.2.3; Section 6.2.3 Step 2.1.1.3; and Section 7.2.1 Step 1.3]. There are no default initial values for variables. For variables without the initial attribute, the standard generates variables with an "undefined" initial value [2, Section 7.2.9 Step 1.2]. These variables have certain contexts in which they "must not" be "undefined" [2, Section 7.5.2.1]. However, the interpretation of a "must" or "must not" test is that any failure of these tests means that the program interpretation is no longer governed by the standard

-- i.e., any action, including ignoring the condition, is allowed [2, Section 1.2(1)]. Although the standard provides no explicit provision for testing for variable initialization, such testing is implicit in the "must not" condition.

c. Testing at run-time for variable initialization is extremely costly unless supported in the hardware. To attempt static compile-time checking would result in compilers of much greater complexity and still achieve only a partial check.

4. Numeric Range and Step Size (D4).

a. Degree of compliance: P

b. Inherent in the precision specification of a fixed point variable is the step size. PL/I ON-conditions allow the testing of subscript ranges [2, Section 7.6.7 Step 4.2] but this does not imply any restriction on the range of a variable; only a restriction on the range of a variable when it is used as subscript to an array.

c. The range of a fixed point variable in PL/I is the entire range of its precision (e.g., with precision (7,3) the range is -9999.999 to +9999.999). The hardware limits of a particular implementation are the only range limits on floating point or integer data.

d. Inclusion of range specifications and runtime range checking would be a large change to PL/I, interacting with facilities such as fixed point arithmetic and parameter passing.

5. Variable Types (D5).

a. Degree of compliance: P

b. PL/I does not allow defined types nor does it allow enumeration types. With these exceptions, there are no restrictions on the structure of data; PL/I allows arrays to be components of records and allows records to be components of arrays.

c. This requirement is wholly in the spirit of PL/I and, were the unavailable types present, compliance would be

total. Type definition is discussed under paragraph E1 below and enumeration types under paragraph E6 below.

6. Pointer Variables (D6).

a. Degree of compliance: P

b. PL/I does provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. A PL/I pointer may point to any form of array or record or to scalar variables of any type, including pointer [2, Section 3.1.19, A170 and A172]. Pointer variables may only point syntactically to variables, records, or arrays explicitly declared as based [2, Section 4.4.5 Case 3.1.(2)]. Any other security checks on pointers and based variables require expensive runtime overhead. For example, the data object dynamically referenced by a pointer variable is required to have certain attributes in common with the data object syntactically pointed to [2, Section 7.6.3]; testing for this match is expensive.

c. Because based variables, records and arrays are explicitly allocated by the program, the scope of allocation is not statically determinable; they may be allocated at any point within their declaration scope [2, Section 7.2.3]. A weakness is that based variables, records, or arrays remain allocated outside their declaration scope unless explicitly freed.

d. The standard does not specify what is to occur when a based variable is "lost" (i.e., when no pointer points to it).

e. The PL/I pointer mechanism was designed before the current knowledge about the impact of language features upon software reliability had been developed. It provides capability without constraint and is basically a hardware-oriented mechanism. It would be almost impossible to impose security on the current PL/I pointer and based storage mechanism. Any major redesign of this mechanism would have effects throughout the language and compilers for the language.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

a. Degree of compliance: P

b. PL/I does not offer a programmer the ability to define new data types or new operators, infix or prefix, in his programs. Data definition in PL/I is limited to specifying the structure of objects; the power and added safety of a type definition facility are not present.

c. The PL/I procedure defining capability is flexible and general, particularly the generic facility, which is useful in many application areas.

d. A type definition capability using solely procedures, dynamic and static storage allocation schemes with built-in data types and structuring is already available. Extension of existing operators to new types (e.g., a type which invokes a procedure when a data object of the defined type is the target of an assignment) would require substantial changes to the design of PL/I.

2. Consistent Use of Types (E2).

a. Degree of compliance: F

b. PL/I has no type definition facility. If a type definition facility were to be added to PL/I then this requirement could be included among the design goals.

3. No Default Declarations (E3).

a. Degree of compliance: F

b. The type (i.e., PL/I attributes) of any program component not explicitly declared by the programmer will be given by default [2, Section 4.3.6.3].

c. As discussed above in connection with A1, restrictions on default definitions should be straightforward to implement but are antithetical to some of the original PL/I design goals.

4. Can Extend Existing Operators (E4).

a. Degree of compliance: F

b. PL/I defines the effect of the existing operators with all valid data types (e.g., arithmetic values may be added or compared; pointers or labels only compared) [2, Section 9.3.2ff].

c. Although not under the user's control, many operators are extended to include the record type. When two records of dissimilar form are operands of an infix operator, an intermediate record is constructed using a complex set of rules. If this can be done the two records are called "compatible" [2, Section 9.1.1.4]. Both records are "promotable" to the intermediate record [2, Section 7.5.3.1]. All infix operations may involve automatic type conversions, again not under user control.

d. PL/I offers no facility for defining new atomic data types and no facility for user controlled extension of existing operators.

e. The extension of existing operators to newly defined data types must be included with the design of an entire type definition task, and provision of an operator extension facility makes this even more difficult.

5. Type Definitions (E5).

a. Degree of compliance: F

b. PL/I currently provides none of the features required in E5. The inclusion in PL/I of data type definition features is a large task that would involve redesign of substantial parts of the language.

6. Data Defining Mechanisms (E6).

a. Degree of compliance: P

b. PL/I offers no facility for defining types by enumeration. PL/I offers arrays of records and records of arrays nested to any depth, and the components may be any PL/I type (e.g., fixed or floating arithmetic variables, character or bit strings, pointer or label variables). PL/I

offers neither discriminated union nor a power set facility, but PL/I does provide bit strings.

c. Different methods for the definition of enumeration types have been proposed and implemented. The simplest of these methods, using a unique literal name to specify a small integer constant in restricted contexts, could be implemented without great difficulty. Power sets of enumeration types may be easily implemented when an enumeration type definition mechanism is available.

d. Discriminated union is discussed under paragraph A7 above.

7. No Free Union or Subset Types (E7).

a. Degree of compliance: P

b. The DEFINE facility of PL/I provides free union. PL/I does not support subsetting. See paragraph A7 above for discussion of the DEFINE facility.

8. Type Initialization (E8).

a. Degree of compliance: F

b. The user of PL/I cannot define new types. Inclusion of initialization and finalization procedures as part of the type definition should be included in the design of the entire type definition facility.

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1)

a. Degree of compliance: PT

b. PL/I follows the scope rules of ALGOL. Thus a name may be referenced within its declaration block and all blocks nested in that block in which the declaration is not overridden. PL/I offers four allocation schemes:

(1) Automatic. On entry to a block, storage for the program structure is allocated and any requested initialization is performed [2, Section 6.2.3 Case 2.1.1]; on exit from the block all automatic storage allocated for that block is deallocated [2, Section 6.2.4 Step 2]. Automatic variables may be referenced from the allocation block and internally nested blocks in which the variable name has not been redeclared.

(2) Static. Storage is allocated and any requested initialization performed as part of the program initialization [2, Section 5.3.2 Step 4 and Section 5.3.6]. A static variable may be referenced from the block in which it is declared and from all blocks nested in that block in which the variable has not been redeclared. Static storage is equivalent to ALGOL "own" variables.

(3) Controlled. The programmer explicitly allocates controlled storage in the block in which it is declared or in any block nested in that block [2, Section 7.2.2 and Section 3.1.14 A92]. Only the latest allocation of controlled storage may be referenced [2, Section 7.6.1 Case 2.4]; thus, controlled storage is effectively a stack. When the block in which controlled storage is declared terminates, any controlled storage allocated remains allocated [2, Section 6.2.4] and defined and may be referenced when that block is reactivated.

(4) Based. The programmer explicitly allocates based storage in the block in which it is declared or in any block nested in that block [2, Section 7.2.2

and Section 3.1.14 A92]. A pointer together with storage identifier selects the particular allocation of based storage which is to be referenced [2, Section 7.6.2]. All allocations of based storage are simultaneously available. When the block in which based storage is declared terminates any based storage allocated remains allocated [2, Section 6.2.4] and defined and may be referenced when that block is reactivated.

c. Although static, based or controlled storage can remain allocated outside their declaraion scope, they can not be accessed outside their declaration scope. (However, use of pointer variables allows exceptions to this rule.)

d. Because PL/I follows the scope rules of ALGOL, a nested block wishing to restrict access to a declared program structure must re-declare the name.

e. Full compliance with this Tinman requirement would require the restrictions on pointers that are also required by paragraph D6 above. These restrictions could be quite difficult to implement.

2. Limiting Access Scope (F2).

a. Degree of compliance: P

b. The names of data program components are accessible as described in paragraph F1 above. Data declared static or controlled may also be declared external, which gives it an unrestricted scope. Procedure names are known only in the block in which they are defined, unless declared external, which gives them an unrestricted scope. However, redeclaration of access scope via PL/I block structure partially meets requirement F2.

c. Design of the entire type definition facility (see Section VI above) must include the design of the mechanism for restricting access scope.

3. Compile Time Scope Determination (F3).

a. Degree of compliance: P

b. The scope of PL/I identifiers is wholly determinable at compile-time. PL/I does not require that identifiers be declared but when they are declared they may be declared anywhere within their scope. Keywords and variable names may be the same with only a contextual distinction. (E.g., 'DO DO=1;' is legal. The second 'DO' is the loop control variable.) Access scopes in PL/I are lexically embedded with the most local definition applying when the same identifier appears at several levels [2, Section 4.4.5].

c. The modifications needed to bring PL/I into compliance with F3 include the restrictions on default declarations (see A1 above) and the requirement that all declarations must appear at the beginning of a block (i.e., before the first executable statement). These are not major changes to the language, but modifying existing implementations may be difficult.

4. Libraries Available (F4).

a. Degree of compliance: PU

b. PL/I offers many mathematical and string handling functions [2, Section 9.4.4].

c. Because of the method used in [2] to define the language, the creation and use of object or compile-time libraries is outside the scope of the PL/I standard. For example, [2, Section 1.4.3.2 Step 2.1] states: "Obtain, from a source outside this definition, a sequence of characters composing a putative PL/I external procedure ..." The PL/I standard, which attempts absolute rigor and unambiguity, could be usefully supplemented by some sort of a "PL/I System Implementers Guide" describing in greater detail features which must be available, as, for instance, libraries.

5. Library Contents (F5).

a. Degree of compliance: PU

b. PL/I has an implementation-defined %INCLUDE facility [2, Section 1.2.2 (8) and Section 2.5.7] which could be used to provide compile-time accessible libraries. This feature cannot be evaluated for this study because the standard does not define what its effect is to be.

c. As in the preceding paragraph (F4), PL/I's conformance with this requirement could better be assessed with a different style of documentation. It is certainly the intention of the standards committee that compile-time source libraries be available. Compile-time libraries containing PL/I source code and object-time libraries containing both PL/I object procedures and object routines whose bodies are written in other source languages are available in existing commercial PL/I systems. Finding a suitably rigorous definition is the only difficulty with inclusion of libraries in the standard.

6. Libraries and Compools Indistinguishable (F6).

a. Degree of compliance: FU

b. The comments under paragraph F5 above apply here also. The inclusion of uncompiled source code, provided by %INCLUDE, offers much lower security than a partially compiled COMPOOL, because the included source text can acquire different attributes from different contexts while the partially compiled COMPOOL will have the same attributes in all contexts.

7. Standard Library Definitions (F7).

a. Degree of compliance: P

b. PL/I offers a general I/O capability with stream (character oriented) and record (block oriented) files. Record files may be sequential or direct (random), keyed or not keyed; the length of a key is implementation-dependent [2, Section 1.2.2 (25)].

c. The ENVIRONMENT attribute allows the programmer to define machine-dependent features in an implementation-dependent manner [2, Section 1.2.2 (5) and (2)]. The syntax, semantics, and any consistency checks of the environment attribute are entirely implementation-dependent.

d. The special machine-dependent features of peripheral equipment are localized in the environment attribute, and the I/O capabilities of PL/I are general enough to handle the special cases.

e. Definition of completely machine independent interfaces to special machine features is now beyond the state-of-the-art. For PL/I, it would require definition of the entire operating system.

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Degree of compliance: P

b. PL/I provides structured control mechanisms for sequential, conditional, iterative, and recursive control, and for exception handling. PL/I does not provide control structures for (pseudo or real) parallel processing or asynchronous interrupt handling.

c. The PL/I "DO" allows multiple termination conditions [2, Section 3.1.10 A64 and A65], but these conditions are tested before each iteration [2, Section 6.3.4 and subsections]. Some iterative forms of the PL/I "DO" require no loop control variable [2, Section 3.1.10 A71].

d. The IBM versions of PL/I have a tasking or parallel processing facility. A similar capability was not included in the standard because it was difficult to define rigorously and it depended heavily on the operating system available. The same can be said of language features to handle asynchronous interrupts. These features would be difficult to define and implement and would substantially increase the size and complexity of an implementation.

2. The GoTo (G2).

a. Degree of compliance: P

b. PL/I provides a "GoTo" operation applicable to, but not restricted to, program labels within the most local scope of definition [2, Section 6.3.7]. PL/I imposes no unnecessay cost for the presence or use of "GoTo". Necessary costs are imposed for the orderly termination of all active blocks when the "GoTo" exits from one or more blocks [2, Section 6.3.7 Case 2.2].

c. PL/I offers label variables and arrays of label constants or label variables which may act as switches, and label parameters; numeric labels are not offered. The PL/I "GoTo" has very low security.

d. As in many other parts of PL/I, the major difficulty with "GoTo" is the provision of unwanted generality.

Removal of label variables and label parameters from the language, which could easily be achieved, would meet many of the goals of this Tinman requirement. More difficult would be the restriction of label namescope to a single level; this requires that treatment of labels be different from the treatment for other names, which could be a major change to identifer processing. Restrictions on the use of arrays of label constants (PL/I's version of computed GoTo) or provision of a case construct (see paragraph G3) would bring PL/I into full compliance with this requirement. It may be noted that PL/I does not have any non-normal loop exit except GoTo.

3. Conditional Control (G3).

a. Degree of compliance: P

b. PL/I conditional control structures are not fully partitioned. When there are nested IF-statements only some of which contain ELSE's, then an ELSE is associated with the "nearest" IF not having an ELSE.

c. The use of an array of label constants allows computed choice among labeled alternaives. This PL/I mechanism -- e.g., "GoTo CHOICES(N);" -- does not offer the clarity or security of the CASE construct.

d. The selection condition in an IF-statement may only be based on a Boolean expression. There is nothing quite comparable to Zahn's device offered by PL/I but the available control structures could be composed to provide a similar capability (though without the security offered by Zahn's device).

e. Restriction and modification of the GoTo and array of label constants mechanism could easily provide a CASE type conditional structure. To extend this CASE structure to discriminated union requires first a discriminated union capability, a large task which was discussed in paragraph A7 above.

f. Only a small change in syntax is necessary to require fully partitioned IF-THEN-ELSE; however, the possibly large number of do-nothing ELSE-clauses may detract from clarity.

4. Iterative Control (G4).

a. Degree of compliance: P

b. The PL/I iterative control structure, the "DO" loop, permits the termination condition to appear only at the beginning of the loop [2, Section 2.3.3 CH6 and Section 2.4.8 CM62]. The control variable is not local to the loop; moreover, there is no requirement that it be local to the block immediately containing the DO-loop. One form of the PL/I DO-loop, the "DO WHILE (expression)", requires no loop control variable. The PL/I loop "must" only be entered from the head of the loop [2, Section 6.3.7.1 Step 1]. (As with all requirements stated in [2], the action when a "must" constraint is violated is implementation-dependent.)

c. The PL/I DO-loop may take several different forms. In all the possible cases, the loop control variable has a well-defined value, but the multitude of cases means that a programmer would have great difficulty determining what that well-defined value is defined to be.

d. Any computed value may be passed outside the loop by assignment to a variable.

e. The addition of a loop exit command that can appear in a THEN or ELSE-clause, a minor addition, would permit the loop termination condition to appear anywhere; GOTO is used in current PL/I compilers for the same purpose. The restrictions on GOTO discussed under paragraph G2 above would allow compile-time checks against jumping into a loop body which is an implementation problem of some difficulty. Restricting the scope of the loop control variable would require some changes to the structure of DO-loops and declarations; the loop control must be implicitly declared and the loop treated as a scope. This is not a minor change.

5. Routines (G5).

a. Degree of compliance: PT

b. PL/I offers recursive routines as a programmer-specified option. There are no restrictions on defining procedures within recursive procedures; including such restrictions is not a major change.

6. Parallel Processing (G6).

a. Degree of compliance: F

b. PL/I offers no parallel processing capability (although such facilities have been included in previous versions of the language).

c. The inclusion of a parallel processing facility in PL/I would be difficult in both the definition and the implementation phases. As stated by Marcotty [15, p. 133]:

> During the course of the development of the language for standardization, a tasking facility was designed and it was only during the final stages of preparation of the definition document that the facility was removed. The main reasons for this action were difficulties with the rigorous definition of the facilities and their interaction with storage. There was also a feeling that with current operating systems, there was no great advantage to be obtained by multitasking within a program. I believe it is currently beyond the state of the art to provide a rigorously defined tasking facility that can be implemented securely and efficiently.

7. Exception Handling (G7).

a. Degree of compliance: P

b. The PL/I exception handling control structure, the on-unit, allows the user to enable transfer of control for any error or exception situation recognized by the system. Aside from conversion errors (e.g., the attempt to convert non-numeric character data to a numeric value) and certain I/O errors, the exception handling mechanism is not parameterized. The exception handling process has no way of determining the source of the exception (i.e., the statement whose execution caused the exception) and, except for conversion errors, no way of correcting the situation that caused the exception.

c. A PL/I exception causes control to transfer backward because a condition is enabled where the procedure for handling the exception is defined. For every exception

condition not explicitly enabled there is a system exception procedure -- implementation defined. The PL/I user can write programs which can escape from an arbitrary nest of control and the exception handling procedure can be redefined for different scopes and for different portions of the same scope.

d. PL/I also provides the user with a mechanism to define his own named conditions which are activated through use of SIGNAL statement [2, Section 6.4.1.3] (e.g., IF I>50 THEN SIGNAL INDEX_TOO_LARGE; where INDEX_TOO_LARGE is a programmer-named condition, which is defined by the programmer-written procedure handling it).

e. While PL/I meets much of this requirement, full compliance would involve great expense and a new definition of this language feature. For instance, parameterizing the recovery requires that all possible exception points and exception types have some mechanism for using this recovery information. To meet the partial requirement that transfers of control be forward in the program would likely result in a facility which is less clear and less useful than the one presently in PL/I.

8. Synchronization and Real Time (G8).

a. Degree of compliance: P

b. PL/I offers no real time or parallel processing facilities; thus, no synchronization features are available. Addition of such a capability to PL/I would involve great expense; see the discussion in connection with G6 above. (It may be noted that previous versions and implementations of PL/I have contained synchronization and real-time facilities which meet some of the requirements of G8.)

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

   a. Degree of compliance: P

   b. The PL/I source language uses free format with an explicit statement terminator [2, Section 2.4 Middle Level Syntax]. PL/I allows the use of mnemonically significant identifiers. Implementations must allow identifiers up to 31 characters in length and may allow larger identifiers. PL/I is based on conventional forms. PL/I does not have a simple grammar. The very large number of keywords (188, which are not reserved), the number of optional forms for nearly every statement type, and the complexity of certain constructs make the language difficult to parse. There are also 87 built-in functions whose names should, in a disciplined programming environment, be reserved unless the user's explicit intention is to replace the built-in function with a programmer-written function serving the same purpose. Abbreviation of many of the keywords is permitted.

   c. Although it is certainly possible to write readable, maintainable programs in PL/I, many aspects of the language design foil this goal. One of the best illustrations of this is the policy of trying to construe a legitimate interpretation for programs which are almost certainly in error. For example, the fragment "DO I=0, I=1; loop body; END" will execute the loop body twice, but each time with the control variable (I) equal to zero because the "I=1" is interpreted as a Boolean expression.

   d. Some of the simplifications for PL/I discussed elsewhere will make programs written in the language easier to read and to compile. The removal of some or all of the unneeded features listed in Section XV would also tend to make PL/I a simpler language. Abbreviated keywords in PL/I are considered fully equivalent to the unabbreviated keywords; based on frequency of usage or some other criterion, only one of the forms should be used. This change would remove 45 keywords, some very unclear (e.g., NOUFL for NOUNDERFLOW or CTL for CONTROLLED), from the language. Reserving particular symbols and operators to a single purpose, as discussed under paragraph H10 below, will also remove much ambiguity and error-proneness from PL/I. All of these changes would simplify new implementations of

PL/I. The changes to existing compilers, while individually quite easy to design and implement, would involve substantial effort because of the multitude of changes required.

2. No Syntax Extension (H2).

a. Degree of compliance: T

b. PL/I offers none of the interdicted features.

3. Source Character Set (H3).

a. Degree of compliance: PT

b. The only PL/I characters not in the 64-character USASCII subset are the "not"-symbol and the "|" (logical "or" and string concatenate) symbols. The language definition does not specify a translation of those two characters.

c. The original versions of PL/I were defined to use the EBCDIC character subset available on keypunches. The two characters above could be translated to circumflex and exclamation point, which have no other PL/I meanings, in the 64 character USASCII subset. These are minor changes.

4. Identifiers and Literals (H4).

a. Degree of compliance: P

b. The PL/I language definition provides formation rules for identifiers [2, Section 2.5.3 CL8] and literals [2, Sections 2.5.4 and 2.5.5]. The identifiers may contain an explicit break character (the underline character). Neither break characters nor blanks may appear in literals. PL/I does not require (or allow) a separate quoting of each line of a long literal.

c. The formation rules for literals could probably be modified to accept (i.e., ignore) embedded break characters. A somewhat larger change would be involved in allowing the separate quoting of long literals. It may be noted that if constant expresions are evaluated at compile time (see paragraph C4 above) then the concatenate operator can be used between the quoted literals with no change to the language syntax.

5. Lexical Units and Lines (H5).

a. Degree of compliance: P

b. PL/I does not treat end-of-line as a delimiter [2, Section 2.5.1 CL3]; thus, lexical units may continue across line boundaries. End-of-line may be included in string literals [2, Section 2.5.5 CL25].

c. Identifiers and strings may not consist of more than one lexical unit nor are constant expressions necessarily evaluated at compile-time.

d. PL/I could easily be changed so that lexical units are not allowed to continue across lines. Constant expressions could be evaluated before run-time. PL/I could only with difficulty allow identifiers to consist of more than one lexical unit.

6. Key Words (H6).

a. Degree of compliance: P

b. PL/I keywords are not reserved (hence are usable as identifiers), are very numerous, and are generally informative. The names of built-in functions of PL/I are treated as identifiers. Program specified condition names and the built-in conditions may appear in the same contexts.

c. Elimination of abbreviations would remove some 45 keywords (i.e., more keywords than exist in FORTRAN) from PL/I. Some keywords could be removed if unneeded features were removed (see Section XV). The remaining keywords of PL/I could be made reserved words in the language. PL/I must continue to have a large number of keywords simply to name the plethora of possible identifier attributes (currently 49 keywords) and exception conditions (currently 29 keywords). Achieving PL/I's goals with only a small set of keywords would be extremely difficult, but the keywords that exist could easily be reserved.

7. Comment Conventions (H7).

a. Degree of compliance: PT

b. PL/I has a single uniform comment convention [15, Section 2.5.2]. A PL/I comment is opened by "/*", closed by "*/". A comment may appear anywhere a blank may appear (i.e., not internally to a lexical unit). A PL/I comment does not terminate at an end of line; only the comment closing symbol terminates a comment.

c. A currently unused character (e.g., "#") could be used to begin a comment terminated by end of line, which would allow PL/I to meet this Tinman requirement without difficulty.

8. Unmatched Parentheses (H8).

a. Degree of compliance: F

b. PL/I allows more BEGINs than matching ENDs and more DOs than matching ENDs, but only when particularly sought by the programmer.

c. The PL/I convention allows a named END statement to close all unclosed BEGIN blocks and DO groups contained within the BEGIN block or DO group with the same name. Elimination of this convention from PL/I would result in no lessened capability in the language, could be easily carried out, and would simplify new compilers.

9. Uniform Referent Notation (H9).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement, since array referencing and function invocations share some similar syntactic forms. However, the "*" form of subscript may not be used as a function argument; except for the "pseudovariables", function invocations cannot appear on the left side of assignment statements while array references can; the dot qualification form for STRUCTURE component selection cannot be used in a function call.

c. Providing an entirely uniform referent notation presents an extremely difficult definitional problem. What would a "*" form of procedure argument mean? What does a function as target of an assignment statement mean? The only approach likely to be successful is the restriction of certain capabilities in the language.

10. Consistency of Meaning (H10).

a. Degree of compliance: P

b. PL/I does allow language defined symbols to have different meanings in different contexts as follows:

(1) Colon ":" delimits statement labels and condition prefixes and separates lower and upper bounds in array dimension declarations.

(2) Period "." serves as the decimal point in literals and a qualifier delimiter for STRUCTURE component selection.

(3) Asterisk "*" is the multiply operator and can stand for an unknown string length or free array bound.

(4) The keyword ERROR can indicate a compile-time attribute error in the DEFAULT statement or serve as the run time exception condition "of-last-resort".

(5) Parentheses "()" either enclose lists or specify precedence.

(6) The equal sign "=" is the assignment operator and a relational operator.

c. Only the last two of the above exceptions to this Tinman requirement are ever likely to lead to ambiguity in practice; item (6) is especially bad (e.g., "I=I=1;" does not have an obvious meaning; see paragraph H1 above.). A new operator for assignment (possibly ":=" which is commonly used for assignment) could be defined. The different functions of parentheses in (5) above result in the different meanings of "()" in CALL PROC(A, (B), C); assuming argument and parameter types match, A and C are passed by reference and B is passed by value. The binding of argument to formal parameter should be specified by the procedure called rather than by the caller. See paragraph C7 above.

Section X. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

a. Degree of compliance: F

b. PL/I implementation may define many parameters: the default and maximum precisions of variables, the size of a data record on an external medium, and many others. An implementation may also define when exception conditions are raised.

c. The behavior of a program which does not conform to the standard is entirely implementation-dependent [2, Section 1.2 ("Relationships between an implementation and This Definition")]:

> An implementation's interpretation of a program-run conforms to the standard if and only if it conforms to one of the conceptual interpretations as follows:
>
> (1) If the conceptual interpretation rejects a program run (via failure of a "must" test) or if it never completes the translation-phase, then any interpretation by the implementation conforms. In particular, an implementation may or may not reject a program-run at the same point as it is rejected by the standard, or at all.
>
> (2) Otherwise, the implementation's interpretation conforms if it makes the same sequence of changes to datasets as does the conceptual interpretation.
>
> (3) The implementation's interpretation also conforms if it deviates from (1) and (2) only as permitted by the flexibilities of interpretation specified in Section 1.2.1.
>
> Note that this implies that an implementation may provide extensions beyond the language defined in this standard, but is still required to conform for a program not using those extensions just as if the extensions were not available.

d. The greatest weakness of PL/I with respect to this Tinman requirement involves the completely undefined

behavior of erroneous programs. In terms of the standard this problem could be solved by defining standandized behavior for certain classes of error. For example:

(1) All syntax errors will generate a message at the earliest possible point (which depends on the parsing technique and the grammar). An object program compiled from source text containing syntax errors shall be executable only up to the point of the error.

(2) There shall be a compiler/system option to detect and announce references to undefined variables.

(3) Jumps into the bodies of loops shall be detected and announced.

(4) Violation of any "must" constraint on the use of pointers shall be detected and announced.

e. Certain of the implementation-dependent features that remain after specifying the behavior of illegal programs are not violations of the Tinman (for instance, allowing the maximum precision to be implementation-dependent). The elimination of defaults throughout PL/I removes the implementation-dependencies in the default specifications. Taken together, the restrictions proposed above involve a definition phase and, depending on hardware support facilities available, a possibly very expensive implementation phase.

2. Object Representation Specifications Optional (I2).

a. Degree of compliance: P

b. PL/I partially satisfies this requirement. The programmer may specify alignment for data objects, thereby overriding the translator-defined default. However, there is no facility for specifying whether a subroutine is to be compiled open or closed. As no parallel processing is defined for PL/I, reentrant vs. nonreentrant is not an applicable option.

c. Although compile-time macro capabilities are not defined in the PL/I standard, many compilers for PL/I support such a feature, which allows a possible

implementation of open subroutines. At the expense of somewhat greater complexity, "open" vs. "closed" could be specified in the procedure heading.

3. Compile Time Variables (I3).

a. Degree of compliance: F

b. PL/I has no features for meeting this requirement.

c. A set of reserved words similar to the set used in JOVIAL (see Chapter 5) could be made available for compile-time and run-time interrogation. The JOVIAL set specifies: bits in a word, bits in a byte, maximum available memory size, etc. Bringing PL/I into compliance with this requirement is a relatively minor modification, both to language and implementation.

4. Conditional Compilation (I4).

a. Degree of compliance: F

b. PL/I provides no conditional compilation facilities.

c. Although not defined in the PL/I standard, a compile-time macro facility is provided in many existing implementations of PL/I. This facility was omitted from the standard because of an inability to agree on a suitably powerful capability and the difficulty of rigorous definition of the interaction with other PL/I capabilities. A facility adequate to meet this Tinman requirement could be defined and implemented at a reasonable cost.

5. Simple Base Language (I5).

a. Degree of compliance: F

b. PL/I provides no extension facilities, so the entire language must be considered the base. Some features of PL/I are redundant, for example: "DO v=e1 BY e2 TO e3;" is a simplication of "DO v=e1 REPEAT e2 WHILE (e3);". Some other features are so similar that it may be difficult to determine which is best for a particular application: CONTROLLED storage and BASED storage both have dynamic allocation under explicit user commands; CONTROLLED storage may be declared external and BASED storage may not; pointers and offsets also have many similarities.

c. It is not realistic to expect to modify PL/I to bring it into compliance with I5 and still have the language retain its essential structure. Pl/I was not designed to be a simple language.

6. Translator Restrictions (I6).

a. Degree of compliance: F

b. These restrictions are a function of the translator implementation [2, Section 1.2.1.2 (1) and (2)]. The language definition does not restrict the number of array dimensions or the number of arguments for a function or subroutine call; when an implementation sets a maximum length on identifiers, that maximum may not be less than 31 characters [2, Section 1.2.1.2 (1)]; the number of nested parentheses levels in expressions is not restricted nor is the number of identifiers.

c. It is a minor modification to add to the language definition a set of restrictions such as the ones called for in I6. The problem lies in deciding what these restrictions should be.

7. Object Machine Restrictions (I7).

a. Degree of compliance: T

b. PL/I satisfies this requirement. Any excessive static requirements for system resources may be announced at compile time or load time; any excessive dynamic requirements for system resources raise run-time exception conditions. No restrictions dependent on the object machine are built into the language definition.

Section XI. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

a. Degree of compliance: P

b. There are many PL/I features which require run-time support even when not used. Strings and arrays require dope vectors, even when they are of fixed length or have fixed dimensions, because argument passing allows run-time determination of parameter bounds. The use of array slices which PL/I allows also has a high run-time cost. Although not strictly a run-time cost, assigning all the defaults to PL/I variables makes the compilers more expensive and slower running. The large number of attributes and the complicated way they are assigned to literals, for example, means that unless a programmer has been extremely careful some implicit conversions will be involved. For example:

```
DECLARE X FLOAT;
X = X + 1.Ø; /* 1.Ø REQUIRES CONVERSION FROM
FIXED TO FLOAT*/
X = X + 1.OEO; /* REQUIRES NO CONVERSION */
```

While this could be optimized, it would require additional overhead for the compiler.

c. The very richness of PL/I encourages programmers to use inefficient features which they do not need and might otherwise avoid. An example is the complicated processing allowed on records (i.e., PL/I structures) of dissimilar form:

```
DECLARE 1 A,
          2 B,
            3 C FLOAT,
            3 D FLOAT,
          2 E FLOAT;
DECLARE  1 Z,
          2 Y(3) FLOAT,
          2 X    FLOAT;
```

The operation A + Z results in an intermediate structure of the following form:

```
1 tmp,
   2 B_Y(3),
      3 C FLOAT,
      3 D FLOAT,
   2 E_X FLOAT;
```

The values are:

```
tmp.B_Y(1).C = A.B.C + Z.Y(1);
tmp.B_Y(1).D = A.B.D + Z.Y(1);
tmp.B_Y(2).C = A.B.C + Z.Y(2);
tmp.B_Y(2).D = A.B.D + Z.Y(2);
tmp.B_Y(3).C = A.B.C + Z.Y(3);
tmp.B_Y(3).D = A.B.D + Z.Y(3);
tmp.E_X = A.E + Z.X;
```

d. The PL/I standard [2, Section 6.3.7.1 Step 1] states one "must not" GOTO a statement within a loop. Because PL/I allows label variables, the compiler cannot check that this error will not occur. Either a high run-time overhead, both of time and information, is required, or the error is ignored.

e. Changes suggested elsewhere in this chapter will contribute to the goal of efficient object code by eliminating implicit conversions, by eliminating the promoting of structures to different forms, and by removing the need for certain run time checks by restricting labels, GOTO and pointer variables. Were all such changes included, PL/I efficiency would compare favorably with other high order languages. However, these are fairly substantial modifications, both to the language and to implementations.

2. Optimizations Do Not Change Program Effect (J2).

a. Degree of compliance: F

b. The PL/I standard explicitly allows certain side effects to be ignored [2, Section 1.2.7.4 "Flexibilities of Interpretation: Expression Evaluation"]. The order of evaluation of contained subexpressions may be changed, which may change the order of evaluation of function subprograms. Any optimizations performed within the language specification could change the effect of a program.

c. The great flexibility allowed by the PL/I standard is explicitly intended to allow compiler optimizations. A restriction that all functions in an expression be evaluated in a left-to-right order reduces the possible optimizations. Certain optimizing compilers would be simpler because of this restriction; nonoptimizing compilers would not be affected. Because of PL/I exception condition processing, a requirement that <u>all</u> side effects occur in left-to-right

order would, for practical purposes, eliminate the possibility of optimization.

3. Machine Language Insertions (J3).

a. Degree of compliance: F

b. PL/I offers no embedded machine language code capability.

c. Implemented PL/I compiler systems allow any procedure, including machine language procedures, with compatible calling sequences to be included in a complete PL/I program. This approach (not formalized in [2]) offers several advantages over embedded assembler code. The interface to the procedure is well-defined, and on another machine another procedure with the same interface could be written; the machine language procedure, once written and tested, could be included in a library and shared by many users, which is not so easy for embedded code; the interface to PL/I is narrower; and PL/I would not need a full target machine assembler to compile programs.

d. If machine language subroutines are not adequate to meet this requirement then a substantial extension to PL/I is necessary. The access rights of the machine language code to the PL/I declared variables, the preservation and restoration of processor states at the entry and exit points of the machine language code and many other such points must be defined. Moreover, a full or nearly full assembly capability must be added to the already large PL/I compiler.

4. Object Representation Specifications (J4).

a. Degree of compliance: P

b. PL/I partially meets this requirement for records by allowing an UNALIGNED attribute (UNALIGNED data are packed as densely as possible). It is possible to specify the order of fields (order in the DECLARE statement), the width of fields, and the presence of "don't care" fields. The ALIGNED attribute will force data to a word boundary. A programmer may specify the precision to minimize the storage required.

c. The PL/I programmer may not explicitly control the object representation of a composite data structure; however, it is possible to tailor alignments and precisions to a particular object machine and so remove portability in a quite insidious fashion.

d. Use of the "compile time variables" required in I3, together with existing features of PL/I such as arbitrary length bit strings and DEFINED data, are adequate to meet this requirement. Something similar to the more sophisticated and explicit strategy used in the JOVIAL specified table would involve a large design and implementation effort.

5. Open and Closed Subroutine Calls (J5).

a. Degree of compliance: F

b. PL/I does not provide the user the option of specifying open vs. closed subroutine invocations.

c. Some PL/I implementations offer a compile-time macro facility, which achieves one form of open subroutine. Defining a procedure to be "open" in the procedure heading would involve extensions to PL/I. The effect on the language is minor, but impact on implementation may not be trivial.

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

a. Degree of compliance: PU

b. While PL/I without question requires the presence of an operating system, the standard uses the hardware/operating system combination as an abstract machine. The standard defines an abstract interpretive PL/I machine and does not distinguish among operating system capabilities, run time support capabilities, and those capabilities compiled directly into the object source code. Substantial operating system capabilities are required to serve the wide I/O capability of PL/I and to provide the memory management required for automatic, based, static, and controlled storage. In a typical PL/I system many of the exception conditions must be processed by the operating system before control passes to the user exception condition process.

c. It is not feasible to bring PL/I into compliance with K1 without thereby conflicting with other Tinman requirements (e.g., B10, G6, G7, G8).

2. Program Assembly (K2).

a. Degree of compliance: PU

b. The PL/I standard permits a program to be composed of separate procedures [2, Section 1.4.3.3 and Section 3.1.1]. The compilation and linking of these separate procedures is described only interpretively and abstractly. The standard validates all procedure interfaces as though the program were produced in a single compilation.

c. Existing PL/I compilers and systems provide separate compilation and libraries, and fully support integration of separately written modules into an operational program. Existing systems generally offer an inadequate check of the interface between separately compiled modules. Since the linker, which should provide the validation, is common to many different compilers, it would be difficult to improve this validation without great expense.

3. Software Development Tools (K3).

a. Degree of compliance: U

b. The PL/I standard defines only the abstract interpretive machine. There is no mention whatever of the desirability of listings, cross references, linkers or any type of debugging system.

c. Existing PL/I implementations provide many of the Tinman-required software development tools, although there is no user interface which is consistent from system to system. An implementer's guide that described, more informally than the PL/I standard, a suggested programming environment would be extremely useful. Possible topics include: listing format and information provided; the format of error messages and when they should appear; and the debugging information which should be available.

4. Translator Options (K4).

a. Degree of compliance: U

b. Comments under paragraph K3 above apply here as well.

5. Assertions and Other Optional Specifications (K5).

a. Degree of compliance: PU

b. Any text having the status of comments can be included as comments. PL/I offers no assertion feature; however, the conditional and output (e.g., PUT DATA) could be used to provide a substitute. If there is a compile-time macro facility available, then nearly all the requested flexibility would become available.

Section XIII. TRANSLATORS

1. No Superset Implementations (L1).

a. Degree of compliance: PU

b. Supersets are explicitly permitted in [2, Section 1.2]: "Note that this implies that an implementation may provide extensions beyond the language defined in this standard, but is still required to conform for a program not using those extensions just as if the extensions were not available."

c. The PL/I standards committee also has a subcommittee investigating a real time version of PL/I. The intention is to design a set of real time tasking features that can be well defined and then add those features to the full language standard during the coming standardization review cycle.

d. The restriction of supersets could be easily incorporated into the PL/I standard definition; however, it is questionable whether supersets would disappear under such pressure. A more effective approach might be to require a compiler option to list all deviations from strict standard conformance; such an option would be straightforward to implement.

2. No Subset Implementations (L2).

a. Degree of compliance: U

b. The PL/I standard has no provision for subsets.

c. PL/I became a USANSI standard language on 9 August 1976, so there are as yet no fully standard versions of PL/I implemented. The size and complexity of PL/I are such that only large machines can support full PL/I systems; thus, it is likely that subset languages will appear. In fact, the PL/I standards committee has formed a subcommittee to define a general purpose subset of PL/I suitable for standardization.

3. Low Cost Translation (L3).

a. Degree of compliance: PU

b. Low cost translation is a feature of the compiler and the system as well as the language. PL/I has a number of difficult-to-compile features, such as the default declaration scheme, many unreserved keywords, and operations on STRUCTURE operands. Significant redesign is needed if low cost translation is to be a serious objective of PL/I.

4. Many Object Machines (L4).

a. Degree of compliance: U

b. This is a management consideration and is not addressed in the language definition.

5. Self-Hosting Not Required (L5).

a. Degree of compliance: U

b. This requirement is a mangement consideration and is not addressed in the language definition.

6. Translator Checking Required (L6).

a. Degree of compliance: U

b. The PL/I standard explicitly makes all error checking dependent upon implementation. As stated in [2, Section 1.2]:

> An implementation's interpretation of a program-run conforms to the standard if and only if it conforms to one of the conceptual interpretations as follows.... If the conceptual interpretation rejects a program-run (via failure of a "must" test) or if it never completes the translation phase, then any interpretation by the implementation conforms. In particular, an implementation may or may not reject a program-run at the same point as it is rejected by the standard, or at all.

c. The PL/I standard could specify classes of errors which are to be detected and, for each class, when they are

to be detected and what is to be done upon detection. This modification would require a significant amount of effort.

7. Diagnostic Messages (L7).

a. Degree of compliance: U

b. The PL/I standard provides no specification or suggestion of useful diagnostic or warning messages. As noted under L6 above, detection and announcement of errors of any type is entirely implementation dependent.

8. Translator Internal Structure (L8).

a. Degree of compliance: T

b. The PL/I standard nowhere specifies what the internal structure of a translator should be.

9. Self-Implementable Language (L9).

a. Degree of compliance: U

b. This requirement involves management considerations not addressed in the standard. PL/I has all capabilities needed to write a compiler. The current lack of type checking is the major shortcoming.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

a. Degree of compliance: P

b. As an existing commercial language PL/I uses only features within the state of the art. Some research would be needed to achieve the Tinman's needed characteristics. This is especially true for the parallel processing and data abstraction capabilities.

2. Unambiguous Definition (M2).

a. Degree of compliance: P

b. The PL/I standard defines the language using a simplification of the Vienna Definition Language. This simplification is a stylized English prose type of programming language, which attempts to be formal and rigorous while remaining more comprehensible than the Vienna Definition Language. Most of [2] can be understood by anyone with a good working knowledge of high-order languages; parts of it are extremely confusing and some ambiguities and "bugs" remain in the document. The committee has improved, and continues to improve, the definition by recomposing the confusing portions and removing ambiguities and bugs. A major flaw in [2] is the total absence of specification for the processing of illegal programs.

c. The PL/I standard comprises the folowing sections: the concrete syntax; the abstract syntax; the translator; the PL/I interpreter (i.e., the PL/I machine); and the flow of control and action routines for the PL/I machine. The concrete syntax describes the form the input program text must take. Thus, all keywords and delimiters are defined in the concrete syntax. The translator describes the translation of the concrete syntax into the abstract syntax, which reflects the concrete syntax but has all the identifier attribute lists completed using defaults, and the entire program represented in a standardized tree form with no keywords or delimiters. The PL/I interpreter describes an idealized processor which runs programs in the tree format of the abstract syntax. This processor has no speed or storage constraints. The flow of control and action

routines describe interpretively, in a stylized English prose, the processing of each PL/I unit from the procedure entry and exit actions to the single expression actions.

d. The PL/I standard is intended for the language implementor rather than for the PL/I user. The user should know the implementation-dependencies as well as any non-conforming parts defined by his implementation before using a PL/I implementation.

3. Language Documentation (M3).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

c. All implementations of PL/I have a set of documentation ranging from the introductory level to quite formal descriptions of the language. Also generally described are the operating system interfaces and other implementation-dependencies. PL/I is used in a number of text books on introductory programming, which provide a machine-independent guide to the language.

4. Control Agent Required (M4).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

5. Support Agent Required (M5).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

6. Library Standards and Support Required (M6).

a. Degree of compliance: U

b. This requirement is a management consideration and is not addressed in the language definition.

Section XV. CONCLUSIONS REGARDING PL/I

1. Conflicts Between the Objectives of PL/I and the Tinman.

a. The major conflicts between the Tinman and PL/I arise because of differences between their respective design goals. Many Tinman requirements involve security and reliability. PL/I, however, was designed primarily for power and for wide applicability. One of the major design goals of PL/I was to include sufficient power and generality to allow any program written in either FORTRAN or COBOL to be better written in PL/I. Modularity (i.e., that a user need know only those PL/I features directly applicable to his problem) and commonality of features with COBOL and FORTRAN were also goals. PL/I includes features from COBOL, FORTRAN, JOVIAL and ALGOL; opinion differs on how well the features of these disparate languages have been integrated into a consistent whole. PL/I is designed to facilitate the initial writing of a program, at the expense of ease in debugging and maintenance. Examples abound: the non-mnemonic abbreviations for exception conditions (e.g., NOUFL for no UNDERFLOW) that will likely occur only once in programs; the labelled end that closes multiple blocks, which even in very large programs cannot save significant effort, and, of course, the defaults and the implicit conversions. PL/I lacks any generalized extensibility and thus the base language supporting all its many features is huge. New features can be included in the base language, at the cost of increased size and complexity; nonetheless, this is the general approach taken by proponents of PL/I. This is understandable in view of the eclectic nature of the current PL/I language.

b. The goals of the Tinman clearly differ those of PL/I. Although power is important in a Tinman-type language, this is secondary to reliability and security. The Tinman favors ease of program debugging and maintenance over ease of writing. It may be noted that many of the Tinman objectives are the direct result of adverse experience in the use of PL/I; thus, unrestricted power in a language is not likely to be a successful approach.

2. Summary of Major Areas of Conflict Between PL/I and Tinman.

a. Data and Types. Although PL/I partially satisfies each of these requirements, PL/I applies much weaker restrictions on defaults and declarations than those the Tinman would apply. Of the required features, PL/I lacks (1) a discriminated union facility, and (2) explicit, namescope-level specification of floating point precision.

b. Operations. The major conflict here is that PL/I allows implicit conversions from any computational mode to any other and that it allows operations on non-conformable records. PL/I also lacks enumeration types and power sets, but many of the effects of these operations may be achieved with bit strings. Certain fixed point operations (e.g., division) in PL/I produce effects somewhat different from those required by the Tinman.

c. Expressions and Parameters. PL/I has a number of serious conflicts with the Tinman in this area. Side effects in PL/I are not constrained to occur in a left-to-right order. The PL/I language definition allows, but does not require, the compile-time evaluation of constant expressions. PL/I does not effectively enforce type agreement in parameters, nor does it have the capability to specify binding class on the formal parameter side. A fixed number of arguments to procedures is required. Most of these conflicts result from PL/I's relative lack of constraints and security enforcement.

d. Variables, Literals and Constants. PL/I has two major conflicts with these Tinman requirements: PL/I has no range restrictions on variables, and the pointer mechanism is not secure. With both of these requirements the changes necessary for conformance to the Tinman are difficult to define well and would involve great expense in implementation and again in run-time overhead. PL/I's other shortcomings in this area could be removed fairly easily. Constant value identifiers could be added to the language; the specifications for numeric literals and input data and their relationship could be made more explicit. Without hardware support it is difficult (i.e., very expensive at run-time) to determine if variables have been initialized.

e. Definition Facilities. In this area, PL/I conflicts with the Tinman in that (1) PL/I lacks any encapsulated data type definition capability, (2) PL/I allows defaults, (3) PL/I does not provide data definition through enumeration of

literal names, and (4) PL/I does not provide for operator extension or definition. Enumeration types could be included in PL/I without great difficulty but the true type definition and data abstration capability which the Tinman seeks would involve a major extension to the language.

f. Scopes and Libraries. Aside from features altogether missing from PL/I because it lacks type definition capabilities, PL/I has no major conflicts with Tinman requirements in this area. The nature of PL/I's defining document is such that many "real world" characteristics, such as libraries, are not really described. The PL/I standard describes the behavior of a program comprising standard conforming procedures irrespective of the source of those procedures. All PL/I systems have libraries but the nature and use of such libraries are not defined in the PL/I standard. The use of %INCLUDE, which inserts source code into the program, is not so secure as a COMPOOL, which uses partially or fully compiled programs. This is because the attributes of identifiers in %INCLUDEd text can depend on the surrounding source (i.e., can vary from procedure to procedure), whereas the attributes of identifiers in a COMPOOL are fixed for all invocations of the COMPOOL.

g. Control Structures. PL/I conflicts with these Tinman requirements primarily in that PL/I lacks parallel processing facilities and has an unrestricted GOTO. Minor conflicts are PL/I's lack of a CASE-type conditional construct and its lack of a loop termination condition that can appear anywhere in a loop.

h. Syntax and Comment Conventions. The single major conflict in this area involves uniform referent notation. PL/I's very large number of keywords also presents difficulties, but the simplifications suggested throughout this chapter will be helpful here.

i. Defaults, Conditional Compilation and Language Restrictions. PL/I has neither provisions for parameterizing the object machine nor any facility for conditional compilation. The most serious conflict with these Tinman requirements lies in PL/I's leaving all error detection and processing entirely implementation-dependent.

j. Efficient Object Representations and Machine Dependencies. PL/I has several major conflicts with Tinman

requirements in this area: PL/I does not contribute effectively to a programmer's writing efficient code and it contains many very inefficient constructs. The specifically unspecified order of expression evaluation means that optimzations can change program effect; machine language insertins are not provided; and the user has no option for specifying open or closed subroutine calls.

k. Program Environment. Some of these Tinman requirements are not language requiremnts *per se* but rather implementation requirements or possibly separate products (e.g., Software Development Tools (K3) and Translator Options (K4)), which can be specified and purchased by DoD. As for language requrements *per se*, the single conflict between PL/I and the Tinman in this area is that PL/I lacks an assertion capability.

l. Translators. PL/I has major conflicts with these Tinman requirements because the processing of erroneous or non-standard conforming programs is not specified. Supersets are explicitly allowed in the standard. Some of the requirements in this area are not directly applicable to a language.

m. Language Definition, Standards and Control. These characteristics do not pertain to the language *per se*; it is not applicable to speak of conflicts between the published PL/I standard and these Tinman requirements.

3. Unnecessary Features in PL/I.

a. Introduction. One of the objectives of this report is to identify language features which are not needed to satisfy (but do not conflict with) the Tinman requirements, with a recomendation as to whether such features should be kept or eliminated. PL/I has a number of features which fall into this category. The following brief sections describe these unnecessary features and make appropriate recomendations.

b. BY NAME OPTION. This PL/I feature is carried over from COBOL's Move CORRESPONDING. The statement "A=B, BY NAME;", where A and B are two dissimilar STRUCTUREs with some component names in common (these components need not have the same attributes), will cause the component from "B" to be stored, after appropriate implicit conversions, in the

component of A that has the same component names. The BY NAME feature is costly to implement, does not appreciably increase the capability of the language, and is error-prone to write and difficult to maintain; we recommend its elimination.

c. Picture Data. This is another carry-over from COBOL. A "picture" represents a numeric data field maintained in character form with certain associated formatting information such as currency indicators. Computation with picture variables is very slow; PL/I has very flexible edited input and output which obviates the need for picture data. For these reasons we recommend the elimination of picture data from PL/I.

d. LIKE-Attribute. Use of the LIKE-attribute (in a STRUCTURE declaration) instructs the compiler to complete the declaration copying the component identifiers and attributes declared in the named STRUCTURE. Because the LIKE-attribute mirrors exactly the user's intentions and contributes to ease of maintenance by requiring that changes be made in only one place to affect all occurences of the same construct, we recommend its retention. Note also that all processing for the LIKE-attribute occurs at compile-time and that it use assures that STRUCTUREs to be used together are conformable.

e. CONTROLLED Storage Allocation. This storage allocation class provides the user with a stack of instances of the data object That is, when a CONTROLLED variable (or aggregate) is ALLOCATED the latest active instance of that variable is pushed and a new top-of-stack is available (and initialized if so declared). When the variable is FREEd then the previous instance is again available. The user could easily simulate this mechanism using based storage or a vector and an index. Because the presence of CONTROLLED storage complicates the language and the memory management system without providing any unique capability, we recommend its elimination.

f. Stream I/O. PL/I has three types of stream I/O: GET/PUT DATA; GET/PUT LIST; and GET/PUT EDIT. GET/PUT EDIT allows the user very explicit and detailed control of the format of the data read or written. For many applications such control is necessary. The LIST and DATA I/O are much simpler to use because the compiler selects the appropriate

formats. However, these two forms overlap. While neither GET/PUT DATA nor GET/PUT LIST are required by the Tinman, we recommend the retention of GET/PUT LIST and the elimination of GET/PUT DATA. GET/PUT LIST provides adequate capability and is much easier to implement.

4. Recommendations Concerning PL/I.

a. On the basis of the evaluation conducted in this chapter, we conclude that an attempt to achieve full compliance of PL/I with the Tinman requirements would necessitate changes so large as to result in a substantially different language. This is because the design objectives of a Tinman-like language and PL/I are so different and also because PL/I was designed before the utility of the requirements captured in the Tinman was known.

b. Some changes that could be easily made to PL/I would improve its performance with regard to Tinman requirements without affecting its power but change the "flavor" of the language. The removal of default declarations and implicit conversions from the language would simplify PL/I and result in better written programs, but would no longer give some meaning to all possibly valid constructs. A PL/I which was changed only by the restrictions described elsewhere in this chapter, would retain the advantages of using an existing language and be a significant improvement over current PL/I.

c. The addition of most Tinman-required features to PL/I would be difficult to do cleanly, would result in a larger and more complex language, and would involve a design, implementation and documentation effort greater than that required for a new language. The language so modified would have little in common with PL/I.

CHAPTER 9

SUMMARY

Section I. EVALUATION TOOL

1. Technical and Overall Evaluation.

One of the objectives of this report is the derivation of a tool which can be used to provide information for the HOL selection process. The tool proposed is a methodology carried out in two stages: a Technical Evaluation (TE) followed by an Overall Evaluation (OE). The TE stage is a conceptually straightforward quantitative approach which is based on the separation of language-dependent and application-dependent elements. The OE stage takes into account factors which are specific to the computing environment in which the language is to be used. This is accomplished via a set of management decision-making criteria, whose values affect the various kinds of costs which comprise the expense for developing a software system. One of these criteria -- in fact, the most important one from the perspective of a language-procurer intending to implement large and long-lived software systems -- is technical merit, the score for which results from the TE stage.

2. Importance of Rating Matrix.

A significant feature of the Technical Evaluation stage is that the central component, the Rating Matrix of language features vs. goals, is independent of both language and application area. Thus, it need be derived only once, rather than for each language. Since this matrix contains almost 500 entries, the fact that it can be used repeatedly facilitates the language evaluation process.

3. Directions for Further Work.

As described in Chapter 2, paragraph I.6, there are several limitations on the applicability of the evaluation tool. First, the very nature of the method (i.e., the attempt to quantify judgments which are essentially imprecise) implies that the numeric results derived will contain a certain amount of "noise". Further research into

techniques to increase the accuracy of such a numeric approach would be desirable to help overcome this limitation. Second, the Overall Evaluation stage tends not to be directly applicable when the language considered in the Technical Evaluation stage differs from actual implementations -- e.g., a new version of an existing language. It is possible that work directed to the issue of estimating the "distance" between languages would be helpful here. Third, the evaluation tool does not reflect the issue of modifying the language to bring it into compliance with the identified requirements. Again, further research is needed in order to so extend the evaluation tool.

Section II. TINMAN REVIEW

1. The "Tinman" as a Requirements Document.

It is important to recognize, both in reviewing the Tinman and in analyzing candidate HOLs with respect to its "needed characteristics", that the Tinman is intended as a specification of language requirements and not as a language design document. For this reason, the presence of conflicts between requirements is not surprising; in fact, it may reflect the state of affairs in the "real world" where, say, efficiency and reliability are uncompromisable objectives. On the other hand, a requirements document must be careful to avoid overspecification. Although the distinction between a requirement and a particular feature which meets the requirement is sometimes subtle, a requirement should not be specified to such a level of detail that solutions which satisfy the intent of the characteristic are ruled out. Unfortunately, the Tinman on occasion falls into this trap (e.g., regarding the pointer mechanism), as cited in Appendix III of this report.

2. Summary of Recommendations regarding the "Tinman".

Detailed comments on the individual characteristics appear in Appendix III. We emphasize here our agreement with the spirit of these requirements and are particularly pleased with the emphasis given to language constructs which promote sound programming methodology. The general comments below suggest changes which we feel would improve the overall quality of the document.

a. Reorganization. A problem with the current Tinman document is that some of the exposition is redundant (e.g., enumeration types arise in A5, B3, B11, D5, and E6). This has the disadvantage that a language lacking such a facility will be penalized several times (during the evaluation) for essentially the same reason. A somewhat different problem with the document is that a single characteristic sometimes covers a variety of requirements (e.g., G1, which includes sub-requirements for an assortment of control structures). This, too, makes the evaluation of a HOL more difficult, since the language may comply with the sub-requirements in varying degrees. Another problem with the Tinman document lies in the occasionally inappropriate section names; e.g., A7, titled "Records", really deals with Discriminated Union

Types; G5, titled "Routines", is concerned with recursion; I1, titled "No Defaults in Program Logic", is basically concerned with implementation dependencies as opposed to defaults.

b. Priorities. Because of the potential and sometimes actual conflicts among the Tinman requirements, it is important for a reader of the document to understand the priorities of the various characteristics. Clearly, the design of a HOL is based on decisions in which tradeoffs were made between conflicting objectives, and it is necessary in evaluating a language against the Tinman to keep in mind the priorities determined by DoD for the requirements of its common language(s). We recommend that such priorities be established, perhaps in the form of weightings for the various characteristics. A possible approach to this subject is provided in the Evaluation Tool described in Chapter 2: the product of the Rating Matrix and Application Vector is a vector whose elements serve, in effect, as weights for the Tinman requirements. (It might be profitable, in view of the wide variety of applications intended to be supported by the common language(s), for several sets of weightings to be determined; e.g., one set emphasizing efficiency at the expense of language power and security, another set sacrificing efficiency where necessary to gain reliability.) There is no question that the establishment of priorities for the Tinman characteristics is a complex task, but unless there is agreement on what is a requirement as opposed to a desirable facility or a luxury which could be omitted, it will be impossible for DoD to assess objectively the language evaluations performed by different contractors.

c. "State-of-the-Art" Features. Although the Tinman requires in M1 that "the language will be composed from features which are within the state of the art", some of the needed characteristics are more in the realm of language research. For example, the notion that a simple kernel coupled with a general set of extension facilities is sufficient to define a "real" programming language (implicit in A2 and E1) has been an elusive goal for researchers in extensible languages for nearly a decade. As another illustration, the issue of data abstraction (E1, E5, E8) is currently an active research area, and the implemented HOLs which provide this facility are in use in academic as opposed to commercial or governmental environments.

3. Recommendation concerning Hardware.

As part of requirement M3, the Tinman states: "The language will be defined as if it were the machine level language of an abstract digital computer." We recommend strongly that DoD follow a logical extension of this concept and undertake the establishment of common language(s) together with common hardware on which the language features can be efficiently implemented. For example, if the hardware provided array bounds checking and tagged memory, then efficiency and reliability need not be traded off in the performance of subscripting and initialization checking. It is worthwhile here to recall the comments of Dijkstra on an earlier version of the Tinman requirements [9, pp. 2, 3]:

> ...in the past, when we used "low level languages" it was considered to be the purpose of our programs to instruct our machines; now, when using "high order languages", we would like to regard it as the purpose of our machines to execute our programs. Run time inefficiency can be viewed as a mismatch between the program as stated and the machinery executing it. The difference between past and present is that in the past the programmer was always blamed for such a mismatch: he should have written a more efficient, more "cunning" program! With the programming discipline acquiring some maturity, with a better understanding of what it means to write a program so that the belief in its correctness can be justified, we tend to accept such a program as "a good program" if matching hardware is thinkable, and if with respect to a given machine [the] aforementioned mismatch then occurs, we now tend to blame that computer as ill-designed, inadequate and unsuitable for proper usage. In such a situation there are only a few true ways out of the dilemma
> 1) accept the mismatch
> 2) continue bit pushing in the old way, with all the known ill effects
> 3) reject the hardware, because it has been identified as inadequate...
> I cannot suggest strongly enough each time to select one of the three ways out of the dilemma, and not to mix them. When the second alternative

"continue bit pushing in the old way, with all the known ill effects" is chosen, let that be an activity with which the HOL project does not concern itself: if it does, the "ill effects" will propagate through the whole system.

Section III. SUITABILITY OF CANDIDATE LANGUAGES

1. Introduction.

In this section we summarize the strengths and weaknesses of this report's six candidate HOLs with respect to the Tinman requirements. The approach presented here is to identify the main objectives which the HOLs attempt to fulfill (e.g., learnability, power, efficiency, reliability, maintainability, transportability) and to contrast these with the goals emphasized in the Tinman characteristics. Software reliability and maintainability are the objectives which come through most strongly in the Tinman; transportability, power, and learnability appear as secondary goals; and efficiency receives somewhat less emphasis than the others. We repeat our agreement with these priorities and point to our comments in paragraph 9.I.3 above for a method of achieving efficiency with a Tinman-like language. A summary of the evaluations is given in Table VII at the end of this chapter.

2. TACPOL.

a. The main attraction of TACPOL is its simplicity. The number of features available in the language is relatively small, and the rules governing the use of these features are generally straightforward. As a result, it is expected that compilers for TACPOL can be efficient. In addition, because of the nature of the facilities provided in TACPOL, efficiency in the run-time code is achievable.

b. One of the benefits of a simple language is learnability, and this goal is achieved fairly successfully through the reference manuals ([3] and [6]). These documents are understandable; [3] provides a formal definition, and [6] serves more as a user's guide. The execution model presented in [3] is particularly helpful in describing the behavior of the various language constructs.

c. Unfortunately, the attractive qualities of TACPOL come at the expense of goals which are equally if not more important for a common HOL. The major problems arise in the areas of security, portability, language expressibility and maintainability.

d. With respect to security, TACPOL has a serious gap in its type checking. For example, the overlaying of data objects of different types is built into "reference" parameter passage, which allows subtle errors to be undetectable by the compiler.

e. Portability was apparently not a goal of TACPOL. For example, the limits on the size of scalars are predicated on the existence of a particular target machine architecture. A more serious problem is that the language abounds in "undefined" conditions (e.g., the result of fixed point overflow, out of bounds subscripts) whose effects are dependent on the implementation. In addition, the absence of floating-point data and formatted I/O are major deficiencies in the language.

f. In attempting to facilitate the generation of efficient code, TACPOL has frequently made restrictions on various features. These restrictions, however, are to the detriment of language expressibility, impairing both writability and readability of programs, and thus increasing the effort involved in program maintenance. To take one example, it is not permitted to WRITE a literal value to a file; instead, this value must first be assigned to a variable. As another example, one may not pass by reference a scalar quantity obtained by subscripting an array; instead, one must pass the entire array.

g. In summary, TACPOL (in its present form) diverges too widely from Tinman requirements, and, because of the scope of modifications needed to bring the language into compliance, such an approach is not practical. Since many of the major differences result from TACPOL's basic design philosophy (not simply from features lacking in the language), it is also recommended that TACPOL not be used as the base for the design of a common language.

3. CS-4.

a. The main strengths of CS-4 are in the areas of program reliability, maintainability, transportability, and language power. Reliability and maintainability were fundamental design goals and are illustrated in such facilities as strong type checking (even across separate compilations), "information hiding" (which prevents the user of a program or data abstraction module from writing

programs which depend upon how the module operates), an exception-handling facility under user control, and protection features for data shared among concurrent processes. Transportability is illustrated most clearly in the arithmetic data types (e.g., the user is required to specify range bounds for integer variables and range and precision for reals) and in the operating system interface (which supports features such as I/O and process management in a standard fashion). The facilities cited in this paragraph also illustrate the power of the language.

b. Weaknesses in CS-4 arise in three areas: language complexity; incompleteness of specification; and needed features which are missing from CS-4. The complexity of the language is due to several factors, the most significant of which is the attempt of the language design to obtain reliability and transportability without sacrificing efficiency. As a result the language includes compiler directives to turn off run-time checking in contexts like array subscripting and union component selection; also, there is an elaborate set of rules concerning the actual precision of real and fraction data. Another cause for the language complexity is that interactions between features sometimes result in "special cases" which have to be described; e.g., the rules constraining the kinds of parameter passage allowed for user-defined implicit procedures in data abstractions. One possible way to eliminate a major cause of these interactions is to remove the data abstraction facility from the language, replacing it with a simpler type definition mechanism such as is found in PASCAL. Although data abstraction directly satisfies some of the Tinman requirements (B1, E1, E5, E8), this is currently a research area among programming language investigators, and many of the advantages of data abstraction can be realized through other CS-4 facilities (most notably, separate compilation).

c. A second weakness in CS-4, mentioned above, that of incompleteness of specification, is apparent in the Operating System Interface and also in the reference manual. It is likely that attempts at a complete specification will reveal opportunities for reducing complexity in the language. The third weakness, needed features missing from CS-4, is illustrated by the absence from the language of pointers, recursion, operator extension, and a compile-time facility such as a macro processor. (It might be noted that

the absence of pointers and recursion from CS-4 does not imply that storage allocation can be carried out completely at compile-time; the presence of run-time determinable array bounds thwarts this goal.)

d. Among the six candidate HOLs considered in this report, CS-4 comes closest to satisfying the Tinman requirements. The aspect of CS-4 which separates it most clearly from the other five HOLs is that constructs which promote sound programming methodology (e.g., type definition and type checking, "structured" control flow facilities) are enforced by the language itself. (Although it is perhaps possible to obtain some of the desirable attributes of such constructs in other languages, with a macro facility and/or a management-imposed programming discipline, it is preferable for the language to contain the enforcement mechanism.) The changes required to bring CS-4 into compliance with the Tinman are minor to moderate in scope and can be made without violating the spirit of the language; these changes are primarily the additions cited in paragraph c above. Thus, CS-4 is suitable, and we recommend its selection, as a basis on which to define a language that satisfies the Tinman requirements.

4. JOVIAL (J73/I).

a. The main strengths of J73/I are, first, that its features are realizable with a fairly high degree of efficiency; second, that its macro (DEFINE) facility makes it possible to overcome some of the language's drawbacks; and third, that it is a successor to a family of HOLs which has been widely used for embedded computer applications (at least in the Air Force).

b. Weaknesses of JOVIAL arise in connection with transportability, simplicity, reliability, readability and power. The fact that machine dependencies permeate the entire language rather than occurring in encapsulated, specialized facilities; the silence of the defining document concerning the detection of errors; and the fact that there is no specification of I/O, process management, and exception handling, combine to defeat the goal of software portability in JOVIAL. Despite its relative smallness, JOVIAL is not an especially simple language; the variety of special cases concerning such facilities as parameter passing, type matching, and data allocation illustrate some

of the complexity. Reliability was sacrificed to obtain run-time (and sometimes compile-time) efficiency; examples are the presence of storage overlaying as opposed to discriminated union, a pointer facility which is insecure, and the absence of type checking when tables or blocks are passed as parameters. Shortcomings with respect to readability are revealed most clearly in the notation for variable declarations, which makes use of a variety of non-obvious abbreviations. The absence of facilities for the definition of data types detracts seriously from the power of the language.

c. Because of the clash between the objectives of JOVIAL and those of the Tinman, JOVIAL fails to satisfy a large number of the Tinman's requirements, and it is not practical to attempt to modify the language to bring it into compliance. For a similar reason, we recommend that JOVIAL not be used as the basis for the design of a common Tinman-oriented language: many of the basic design tenets of JOVIAL would have to be modified, and the resultant language would bear little resemblance to JOVIAL.

5. FORTRAN

a. The main strengths of FORTRAN lie in the efficiency of most of its constructs, its relative simplicity, and its widespread availability (at least for previous versions of the language). In addition, the presence of character data solves one of the chief problems with earlier version of FORTRAN.

b. The language's weaknesses occur in connection with the objectives of power, portability and reliability. The absence of a mechanism for defining heterogeneous data objects (such as PL/I "structures"), the lack of a type definition facility, restriction to static allocation (and thus the absence of a pointer facility and recursion) are major deficiencies. Despite the widespread availability of FORTRAN, there are basic conflicts with portability: floating point data are specified as single or double precision (a machine dependent notion) as opposed to a range of values with an indicated minimum number of digits of precision; also, the reference document leaves to the implementation the decision as to whether a program is legitimate or not. The compromising of program reliability is illustrated in storage-oriented features such as COMMON

and EQUIVALENCE and the lack of type checking in contexts such as passage of array parameters.

c. The comments in paragraphs 2g and 4c above concerning TACPOL and JOVIAL are applicable also to FORTRAN. It is not realistic to attempt to modify FORTRAN to bring it into compliance with the Tinman; we also advise that FORTRAN not be used as the basis for the design of a common language. Programming methodology has progressed considerably in the twenty-or-so years since many of FORTRAN's features first appeared, and a common language should take advantage of these developments.

6. COBOL

a. The main strength of COBOL is that it is a language which is well-matched to its intended application area (business data processing). With its diversity of character-oriented I/O facilities, COBOL is adequate for the specification of simple, transaction-based algorithms. The hierarchical program organization required in COBOL (e.g., the four divisions) coupled with the English-like notation in the PROCEDURE DIVISION, contribute toward program readability. The availability of COBOL (it is the most widely used HOL) is an obvious strength, and we point out also that the design of hardware architectures to support efficiently the constructs of the language (see our recommendations in paragraph 9.I.3 above) has been successfully engineered (e.g., the Burroughs B3700).

b. There are a variety of fundamental weaknesses in COBOL which make it unsuitable for application outside business data processing. First, the data types and data definition facility are inadequate; particular weaknesses are in the areas of arithmetic and boolean data. Second, the control features offered by COBOL are not powerful enough to support general-purpose programming. The ability to invoke subroutines is limited (and distributed between two separate facilities), and there is no block-structure. Third, the language is fairly complex, and many of the rules are plagued by special cases and exceptions, owing to the interactions between language features. Programming practices which tend to be error-prone (e.g., overlaying data via a "free" as opposed to "discriminated" union) are often necessary in COBOL programs. Fourth, there are a variety of implementation dependencies in the language which

defeat the goal of portability. A striking example is the language definition's failure to specify floating-point data and arithmetic.

c. For many of the same reasons given in connection with TACPOL, JOVIAL, and FORTRAN, it is unrealistic to attempt to modify COBOL to bring it into compliance with the Tinman. Moreover, such factors as the language's overriding orientation toward character data and the absence of definitional facilities make COBOL a poor basis for the design of a common language.

7. PL/I

a. The main strengths of PL/I stem from the power of the language and the support provided by various manufacturers (e.g., IBM, Honeywell, and Burroughs). Significant examples of the language's expressive power are its procedure definition facility (e.g., its generic mechanism), its exception handling features, and its variety of I/O and character manipulation constructs.

b. Weaknesses in PL/I emerge in connection with the goals of learnability, efficiency, reliability, maintainability, and (despite the fact that this is also one of the language's strong points) expressive power. PL/I is a large and complex language with features sometimes interacting in obscure ways. The overhead required at run-time to support its constructs (e.g., implicit conversion) is considerable. The widespread appearance of defaults and the insecurity of the pointer facility interfere with reliability and maintainability. The absence of a type-definition mechanism (such as is found in PASCAL) is a serious deficiency and detracts from the power of the language.

c. The substantial differences in philosophy between PL/I and the Tinman imply that an attempt to modify PL/I to bring it into compliance with the Tinman's requirements would amount to a redesign of a large part of the language. This is not a realistic approach. PL/I was based on the idea that a HOL should allow its programmers the widest freedom in using the features of the language. However, recent developments in programming methodology have revealed that <u>security</u> of language constructs, and not unlimited <u>freedom</u>, is desirable in the interests of readability,

reliability, and maintainability of software. Because of these fundamental differences between PL/I and the Tinman, we recommend that PL/I not be used as the basis for the design of a common HOL.

8. Summary of Recommendations.

a. With respect to choosing a HOL either as is or as the base for the design of a common language to satisfy the Tinman requirements, CS-4 is superior to the other five candidate HOLs considered in this report (TACPOL, JOVIAL J73/I, FORTRAN, COBOL and PL/I). This conclusion results from the fact that the high-priority objectives of the Tinman (software reliability, maintainability) are met most closely by CS-4. The application of the Evaluation Tool lends credence to this result. Even if we take into account the inaccuracies to which such a quantitative approach is subject, CS-4 emerged with a significantly higher score for technical merit (approximately 76 out of 100) than the other candidate HOLs (51 for PL/I and JOVIAL, 49 for TACPOL, 47 for COBOL, and 42 for FORTRAN).

b. A second recommendation is that the design of a common hardware architecture be undertaken in conjunction with the HOL effort, to ensure an efficient match between language and machine. The design of a language as powerful as one which would comply with the Tinman will likely fail by virtue of its complexity if it is constrained to be efficiently supportable on the variety of hardware configurations currently in existence within DoD.

c. A third recommendation is that the critiques of the various candidate HOLs be used as input for the respective standards groups, insofar as these reviews suggest improvements within the spirit of the languages.

d. A fourth recommendation is that the Tinman be revised, simplified and clarified in accordance with our comments in Appendix III.

TABLE X. SUMMARY OF LANGUAGE EVALUATIONS (*)

	TACPOL	CS-4	JOVIAL	FORTRAN	COBOL	PL/I	
A01	P L	T	P L	P ML	FP L	P S	A01
A02	P L	P L	P L	P L	P L	PT S	A02
A03	F L	P S	P S	F SM	F SM	P S	A03
A04	P S	P L	F L	F L	T	P ML	A04
A05	P L	P S	F L	F L	P L	P L	A05
A06	P L	PT S	P S	P SL	P L	P S	A06
A07	F L	PT M	P L	F L	P L	F L	A07
B01	P L	PT S	P L	P M	P L	P L	B01
B02	P S	P S	P L	P SM	P L	P S	B02
B03	P M	PT L	PT M	P M	P L	T	B03
B04	P L	T	PT S	PT S	PT SM	T	B04
B05	F LS	T	P L	PT SM	P S	F MS	B05
B06	P M	T	PT S	P S	P S	P SM	B06
B07	F M	P S	F L	F M	F L	P MS	B07
B08	P L	PT M	P L	P LS	F L	F S	B08
B09	P L	T	P L	P SM	PT S	PT L	B09
B10	P L	PT S	F L	P L	PT L	PT L	B10
B11	F M	T	P MS	P SM	F L	P M	B11
C01	F SL	T	F SM	F SL	P SM	F S	C01
C02	PT S	T	PT S	PT S	P S	P S	C02
C03	T	T	T	T	F L	T	C03
C04	F SL	P S	P MS	P SM	F S	P SM	C04
C05	P M	P L	F L	T	T	P M	C05
C06	P M	PT M	P L	P M	F L	P ML	C06
C07	P L	P L	P L	P SM	F L	F ML	C07
C08	P SM	P M	F L	F ML	F M	P M	C08
C09	F L	F M	F M	F ML	F L	F ML	C09

(*) Key for Degree of compliance: "T" means "Totally meets the requirement;" "P" means "Partially meets the requirement;" "F" means "Fails to meet the requirement;" "U" means "Unknown from the available documents if the requirement is satisfied."
Key for Scope of modifications: "S" means "small," "M" means "moderate," "L" means "large." If two entries appear, the first applies to the language and the second to the implementation.
Example: The entry "F SL" for TACPOL and C01 means that TACPOL fails to satisfy C01, and bringing TACPOL into compliance with the requirement has small impact on the language but a large effect on implementations.

TABLE X (continued)

	TACPOL	CS-4	JOVIAL	FORTRAN	COBOL	PL/I	
DØ1	P M	T	P MS	PT S	F M	F SM	DØ1
DØ2	PT S	PT S	P L	P S	T	P SM	DØ2
DØ3	F ML	T	P SM	P MS	F M	P SL	DØ3
DØ4	P M	P L	P M	F L	P L	F L	DØ4
DØ5	P ML	PT L	P L	P L	P L	P SM	DØ5
DØ6	F L	F L	P L	F L	F L	P L	DØ6
EØ1	F L	P L	P L	F L	F L	P L	EØ1
EØ2	F L	P L	F L	F L	F L	F L	EØ2
EØ3	PT S	T	T	PT M	T	F S	EØ3
EØ4	F L	F L	F L	F L	F L	F L	EØ4
EØ5	F L	T	F L	F L	F L	F L	EØ5
EØ6	P M	T	P L	F L	PF L	P L	EØ6
EØ7	P M	PT M	P L	P SM	P L	P L	EØ7
EØ8	F L	T	F L	F L	F L	F L	EØ8
FØ1	P S	T	PT M	P M	FP L	PT M	FØ1
FØ2	F L	T	P S	F ML	F L	P L	FØ2
FØ3	T	T	T	P L	FP L	P S	FØ3
FØ4	P SL	P S	PT S	T	P L	PU MS	FØ4
FØ5	P SL	T	T	F S	PT SM	PU MS	FØ5
FØ6	P SL	T	T	F S	T	FU MS	FØ6
FØ7	P L	PT M	F L	P L	P L	P L	FØ7
GØ1	P L	PT SM	P L	P L	P L	P L	GØ1
GØ2	P S	PT S	P SM	P MS	PT S	F SM	GØ2
GØ3	PT S	P S	PT S	P M	P M	P SM	GØ3
GØ4	P S	PT S	P S	P M	P L	P M	GØ4
GØ5	F SL	F SM	F SM	F SL	F L	PT	GØ5
GØ6	F L	PT S	F L	F L	F L	F L	GØ6
GØ7	P L	PT S	F L	F L	P L	P L	GØ7
GØ8	P L	P M	F L	F L	F L	F L	GØ8
HØ1	PT S	PT S	P MS	P L	P L	P M	HØ1
HØ2	T	T	T	T	T	T	HØ2
HØ3	T	P S	T	T	T	P S	HØ3
HØ4	PT S	PT S	P S	PT S	P S	P S	HØ4
HØ5	PU S	P S	P S	F S	P S	P S	HØ5
HØ6	P S	P S	P S	P S	P LS	F S	HØ6
HØ7	P S	P S	PT S	P S	P S	PT S	HØ7
HØ8	T	T	T	T	T	F S	HØ8
HØ9	P L	P L	P MS	P L	F L	P LM	HØ9
H1Ø	P S	T	P MS	T	P S	P S	H1Ø

TABLE X (continued)

	TACPOL	CS-4	JOVIAL	FORTRAN	COBOL	PL/I	
I01	F L	P M	F LM	F L	F ML	F L	I01
I02	P M	P SM	P SM	F L	P L	P SM	I02
I03	F M	PT S	T	F SM	F S	F S	I03
I04	F M	P L	T	F SM	F M	F MS	I04
I05	PT S	P M	P S	PT S	F L	F LM	I05
I06	P S	P S	P S	P S	P S	F S	I06
I07	P M	T	PT S	T	T	T	I07
J01	PT S	PT M	PT L	PT S	P M	F L	J01
J02	FU SM	TU	F SL	F SL	TU	F S	J02
J03	P M	PT L	P SM	P S	P M	F L	J03
J04	P ML	PT SL	P L	F SL	P L	P L	J04
J05	F S	T	F M	F S	F L	F SM	J05
K01	P L	T	T	F SL	F L	PU	K01
K02	U	P S	T	U S	P L	PU ML	K02
K03	U L	U	U	U	PU	U	K03
K04	U SL	U	U	U	U L	U	K04
K05	F L	FU M	F S	F S	F L	PU	K05
L01	U	U	F LS	F SL	F SL	F S	L01
L02	U	U	U	F SL	F L	T	L02
L03	U S	P L	U	T	U L	PU	L03
L04	U L	U	U	U S	U	U	L04
L05	U	U	U	U	U	U	L05
L06	U	U	U	F SM	U	U	L06
L07	U	U	F L	F SM	U	U	L07
L08	U	U	T	T	U	T	L08
L09	U M	U	T	U	U	U	L09
M01	P L	P M	T	T	P L	P M	M01
M02	P S	P M	F L	P S	P L	P M	M02
M03	U	U	F L	P L	U	U	M03
M04	U	U	U	U	U	U	M04
M05	U	U	U	U	U	U	M05
M06	U	U	U	U	U	U	M06

LITERATURE CITED

REPORTS

1. American National Standards Committee X3J3. Draft Proposed ANS FORTRAN, published in SIGPLAN Notices, Volume 11, Number 3, March 1976.

2. American National Standards Technical Committee X3J1 and European Computer Manufacturers Association Technical Committee TC10. Draft Proposed American National Standard Programming Language PL/I: BASIS/1-12, Document BSR X3.53; February 1975; and Errata for BSR X3.53, Document X3J1/399; January 1976.

3. Litton Systems, Inc., Data Systems Division. CPCEI Specification for Compiler/Assembler for Fire Direction System Artillery AN/GSG-10() (V). Specification EL-CG-00043082C; Document 595909-600C; Contract DAAB07-68-0154; April 1971.

4. Brosgol, Benjamin M., Timothy A. Dreisbach, James L. Felty, Joel R. Lexier, and Gary M. Palter. CS-4 Language Reference Manual; Document IR-130-2; Contract N00123-74-C-0634; Intermetrics, Inc.; October 1975.

5. Grimes, Donald E., and Joel R. Lexier. CS-4 Operating System Interface; Document IR-130-2; Contract N00123-74-C-0634; Intermetrics, Inc.; October 1975.

6. Litton Data Systems, Inc., TACPOL Reference Manual Programming Support System. Document USACSCS-TF-4-1; Contract DAAB07-68-C-0154; June 1975.

7. Brosgol, Benjamin M., and James L. Felty. Language Requirements Report; Document IR-179-2; Contract DAHC26-76-C-0006; Intermetrics, Inc.; January 1977.

8. American National Standards Institute. American National Standard Programming Language COBOL, X3.23-1974.

9. Dijkstra, E.W. "On a language proposal for the Department of Defense," EWD514, September 17, 1975.

10. Robinson, L., M.W. Green, R. E. Shostak, and J. M. Spitzen. The Verification of COBOL Programs. Contract DAHC-04-75-0011; Stanford Research Institute; March 1976.

JOURNAL/MAGAZINE ARTICLES

11. Rothenbuecher, O. H. "The Top 50 Companies in the Data Processing Industry." Datamation, 22 (June 1976) pp. 48-59.

12. Knuth, Donald E. "Structured Programming with Go To Statements." ACM Computing Surveys, 6 (December 1974) pp. 261-301.

13. Reifer, D. J. "The Structured FORTRAN Dilemma." SIGPLAN Notices, 11 (February 1976) pp. 30-32.

GOVERNMENT PUBLICATIONS

14. Air Force Systems Command, Rome Air Development Center. JOVIAL J73/I Specification. July 1976.

15. Marcotty, Michael. "Suitability of PL/I as the Army Programming Language," Centacs Report No. 47, U.S. Army Electronics Command, Fort Monmouth, N.J. June 1975.

16. Department of Defense Requirements for High Order Computer Programming Languages "Tinman"; June 1976.

BOOKS

17. Sammet, Jean E. Programming Languages: History and Fundamentals. Englewood Cliffs, N.J., Prentice-Hall, 1969.

Appendix I

HIGH ORDER LANGUAGE AVAILABILITY QUESTIONNAIRE

Please list each High Order Language being used in your project. For each language, please answer the questions in Parts I and II.

Part I. General Information.

A. What are the main reasons that this HOL was chosen? If there were any alternate languages under consideration, what are the main reasons that none of these was selected?

B. What are the host and target computers for the compiler?

C. Is more than one dialect of the language being used? If so, why?

D. How many programmers are using the language? (Count programmers using the language part time by estimating the fraction of time spent.)

Part II. Availability Data.

This part consists of three sections. In Section A, you are asked to provide scores for various factors comprising the components of the language availability measure. In Section B, you are asked to supply, for each component, relative weightings for the factors constituting that component. In Section C, you are asked to supply relative weightings for the components themselves.

A. Scores for factors of language-availability components.

Three components (compiler availability, quality of language-oriented support tools, quality of documentation) are broken down into factors. For each factor, you are asked to supply a score (between 0 and 10) reflecting the way in which that factor is met at your site.

A fourth component ("other") is listed to account for factors which might not be conveniently included in the other three. If you wish to supply information in this category, please attach a supplementary sheet explaining the factors

you are scoring. In addition, the compiler availability, tools, and documentation components include a factor named "other" so that additional factors in these categories can be entered.

1. Compiler availability.

a. Compile-time diagnostics.

In the score for this facility, take the following issues into account: Are warnings produced for probable errors (such as code which can never be executed)? Does the compiler recover from program errors without introducing bogus ones? Are error messages specific and meaningful? If the language reference manual defines a set of compile-time errors, are all of these detected? Is compile-time data-type checking performed?

Score for compile-time diagnostics:

b. Run-time diagnostics.

Issues to take into account in estimating the score here are: subscript bounds checking, checking for null pointers on dereferencing operations, meaningful error messages, debug facilities (such as statement or procedure tracing, symbolic dumps, breakpoints).

Score for run-time diagnostics:

c. Object code efficiency.

The score here should be relatively high if the compiled code is suitably efficient for the application area. Inversely, if either assembly language must be used or useful features in the language must be avoided because of object code inefficiency, a low score is appropriate.

Score for object code efficiency:

d. Compiler reliability.

A low score here should reflect a situation in which the compiler has contained serious errors, requiring a long time to correct, resulting in delays or other problems for project personnel. If compiler errors have proved to be relatively infrequent and easy to correct, then the score for this goal should be high.

Score for compiler reliability:

e. Compiler smallness and speed

A low score here should reflect a large, slow compiler which has been a source of bottlenecks in the project. A compact, fast compiler should receive a high score for this goal.

Score for compiler smallness and speed:

f. Listings produced by compiler

Take into account here such attributes as: automatic formatting and indenting, ease of finding errors, cross-references, options for listing the object code.

Score for listings produced by compiler:

g. Other

Explain on attached page. Score:

2. Quality of language-oriented support tools

For each of the following tools which is available, evaluate its performance on a 0 to 10 scale (if the tool is not available, rate it as 0).

a. Language-specific text-editor. Score:

b. Verification aids (such as test program generators, static flow analyzer). Score:

c. Program instrumentation aids (such as dynamic statistics-gathering packages). Score:

d. Other (explain on attached sheet). Score:

3. Quality of language documentation

Rate each of the following on a 0 to 10 scale.

a. Language definition document

A high score here should reflect a document which is complete, unambiguous, and readable by an experienced programmer. Deduct points if there have been questions regarding program legality which could not be answered using the document.

Score for language definition document:

b. Programmer's guide

This document is a manual which combines a description of the language (less formal than the definition document)

with a specification of implementation dependencies (e.g., operating-system-oriented facilities, maximum and minimum numeric values). As the main "working document" of the programmer, this manual should be organized to facilitate quick reference but should not sacrifice readability.

Score for programmer's guide:

c. Primer

The primer (possibly a textbook) is an introduction to the language for the novice programmer. Readability is the main goal; the document should contain examples for each feature explained and should also include a set of exercises for the reader.

Score for primer:

d. Other (explain on attached sheet). Score:

4. Other factors related to language availability (explain on attached sheet). Score:

B. Weightings for factors of language-availability composition.

The scores for the various components in Section A above will be weighted by the values which you supply here.

1. Compiler availability

Assign a relative weight to each of the factors associated with the compiler availability. This set of weights should reflect the priorities particular to your project. Each weight should be a fraction (between 0.0 and 1.0), and the sum of the individual weights for the compiler availability factors should be 1.0.

a. Compile-time diagnostics	Weight
b. Run-time diagnostics	Weight
c. Object code efficiency	Weight
d. Compiler reliability	Weight
e. Compiler smallness and speed	Weight
f. Listings produced by compiler	Weight
g. Other	Weight
	Sum = 1.0

2. Quality of language-oriented support tools.

Assign a relative weight to each of the factors associated with language-oriented support tools. This set of weights should reflect the priorities particular to your project. Each weight should be a fraction (between 0.0 and 1.0), and the sum of the individual weights for the support tool factors should be 1.0.

a.	Language-specific text editor	Weight
b.	Verification aids	Weight
c.	Program instrumentation aids	Weight
d.	Other	Weight
	Sum =	1.0

3. Quality of language documentation.

Assign a relative weight to each of the factors associated with language documentation. This set of weights should reflect the priorities particular to your project. Each weight should be a fraction (between 0.0 and 1.0), and the sum of the individual weights for the language documentation factors should be 1.0.

a.	Language definition document	Weight
b.	Programmer's guide	Weight
c.	Primer	Weight
d.	Other	Weight
	Sum =	1.0

4. Other

If you supplied a set of factors with individual scores, assign relative weights to these factors using the same conventions as for components 1 through 3 above.

C. Weightings for the language availability components

The scores supplied in section A and the weightings given in section B will determine a set of four intermediate scores, each between 0 and 10, one for each component (compiler, tools, documentation, other). In the final step in the procedure for deriving a single score for language availability,

assign weights to each of these components. Again, base these weights on the priorities specific to your project. These weights should be between 0.0 and 1.0 and sum to 1.0.

1.	Compiler availability	Weight
2.	Quality of language-oriented support tools	Weight
3.	Quality of language documentation	Weight
4.	Other	Weight
	Sum =	1.0

Appendix II

DEPARTMENT OF DEFENSE
REQUIREMENTS FOR HIGH ORDER
COMPUTER PROGRAMMING LANGUAGES
"TINMAN"

June 1976

Section IV

A. DATA AND TYPES

1. Typed Language
2. Data Types
3. Precision
4. Fixed Point Numbers
5. Character Data
6. Arrays
7. Records

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

By the type of a data object is meant the set of objects themselves, the essential properties of those objects and the set of operations which give access to and take advantage of those properties. The author of any correct program in any programming language must, of course, know the types of all data and variables used in his programs. If the program is to be maintainable, modifiable and comprehensible by someone other than its author, the the types of variables, operations, and expressions should be easily determined from the source program. Type specifications in programs provide the redundancy necessary to verify automatically that the programmer has adhered to his own type conventions. Static type definitions also provide information at compile time necessary for production of efficient object code. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

These are the common data types and type generators of most programming languages and object machines. They are sufficient, when used with a data

definition facility (see E6, D6, and J1), to efficiently mechanize other desired types such as complex or vector.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

This is a specification of what the program needs, not what the hardware provides. Machine independence, in the use of approximate value numbers (usually with floating point representation), can be achieved only if the user can place constraints on the translator and object machine without forcing a specific mechanization of the arithmetic. Precision specifications, as the minimum supported by the object code, provide all the power and guarantees needed by the programmer without unnecessarily constraining the object machine realization. Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program. Specification of the precision will also contribute to the legibility and implementability of programs.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Scaled integers are useful approximations to real numbers when dealing with exact quantity fractional values, when the object machine does not have floating point hardware, and when greater precision is required than is available with the floating point hardware. Integers will also be treated as exact quantities with a step size equal to one.

A5. Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. The definition of a character set should reflect on the manner it is used within a program and not necessarily on the

print representation a particular physical device associates with a bit pattern at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as, USASCII and EBCDIC will be available in a standard library. Note that access to a linear array filled with the characters of an alphabet, A, and indexed by an alphabet, B, will convert strings of characters from B to A.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

This is general enough to permit both arrays which can be allocated at compile or load time and arrays which can be allocated at scope entry, but does not permit dynamic change to the size of constructed arrays. It is sufficient to permit allocation of space pools which the user can manage for allocation of more complex data structures including dynamic arrays. The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This provides all that is safe to use in CMS-2 and JOVIAL OVERLAY and in FORTRAN EQUIVALENCE. It permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

B. OPERATIONS

1. Assignment and Reference
2. Equivalence
3. Relationals
4. Arithmetic Operations
5. Truncation and Rounding
6. Boolean Operations
7. Scalar Operations
8. Type Conversion
9. Changes in Numeric Representation
10. I/O Operations
11. Power Set Operations

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

Equivalence is an essential universal operation which should not be subject to restriction on its use. There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Equivalence meaning logical identity and not bit-by-bit comparison on the internal data representation, however, is required for all data types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of a disjoint types never be identical. Consequently, its usefulness at run time is restricted to data of the same type or to types with nonempty intersections. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

Numbers and types defined by enumeration have an obvious ordering which should be available through relational operations. All six relational operations will be included. It will be possible to inhibit ordering definitions when unordered sets are intended.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

These are the most widely used numeric operations and are available as hardware operations in most machines. Floating point operations will be precise to at least the specified precision.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

These requirements seem obvious, particularly for floating point numbers and yet many of our existing languages truncate the most significant mantissa digits in some mixed and floating point operations.

B6. The built-in Boolean operations will include "and," "or," "not," and "nor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Short circuit mode as used here is a semantic rather than an implementation distinction and means that "and" and "or" are in fact control operations which do not evaluate side effects of their second argument if the value of the first argument is "false" or "true," respectively. Short circuit evaluation has no disadvantages over the corresponding computational operations, sometimes produces faster executing code in languages where the user can rely on the short circuit execution, and improves the clarity and maintainability of programs by permitting expressions such

as, "i<=7 & A[i] >x" which could be erroneous were short circuit execution not intended. Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. In many situations component by component operations are done on array elements. In fact, a primary reason for having arrays is to permit large numbers of similarly treated objects to have a uniform notation. Operations on data aggregates available directly in the source language hide the details of the sequencing and thereby, simplify the program and make more optimizations available. In addition, they permit simultaneous execution on machines with parallel processing hardware. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. A matrix might be defined using an array, but it will not inherit the array operations automatically. Multiplication for matrices would, for example, be unnatural, confusing and inconvenient if the product operator for matrices were interpreted as a component by component operation instead of cross product of corresponding row and column vectors. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

Transfers between arrays or records of identical logical structure are necessary to permit efficient run time conversion from one object representation to another, as might be done when data is packed to reduce peripheral storage requirements and I/O transfer times but need to be unpacked locally to minimize processing costs.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

Implicit type conversions which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can result in unexpected run time overhead.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

Because ranges do not form closed systems, range validation is not possible at compile time (e.g., "I:=I+1" may be a range error). At best, the compiler might point out likely range errors. (This requirement is optional for hardware installations which do not have overflow detection).

B10. The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

Whether the referenced "files" are real or virtual and whether they are hardware devices, I/O channels or logical files depends on the object machine configuration and on the details of its operating system if present. But in any programming system I/O operations ultimately reduce to sending or receiving data and/or control information to a file or to a device controller. These can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write, " with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

As with any data type, power sets will be useful only if there are operations which can create, select and interrogate them. Note that this provides only a very

special class of sets but one which is very useful for computations on sets of indicators, flags, and similar devices in monitoring and control applications. More general sets if desired, must be defined using the type definition facilities.

C. EXPRESSIONS AND PARAMETERS

1. Side Effects
2. Operand Structure
3. Expressions Permitted
4. Constant Expressions
5. Consistent Parameter Rules
6. Type Agreement in Parameters
7. Formal Parameter Kinds
8. Formal Parameter Specifications
9. Variable Numbers of Parameters

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This is a semantic restriction on the evaluation order of arguments to expressions. It provides an explicit rule (i.e., left-to-right) for order of argument evaluation, but allows the implementations to alter the actual order in any way which does not change the effect. This provides the user with a simple rule for determining the effects of interactions among argument evaluations without imposing a strict rule on compilers which are sophisticated enough to detect potential side-effects and optimize through reordering of arguments when the evaluation order does not affect the result. Control operations (e.g., conditional and iterative control structures), of course, must be exceptions to this general rule since control operations are in fact those operations which specify the sequencing and evaluation rules for their arguments.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

This is an example of not imposing arbitrary restrictions and special case rules on the user of the source language. Special mention is made here only because so many languages do restrict the form of expressions. FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The ability to write constant expressions in programs has proven valuable in languages with this capability, particularly with regard to program readability and in avoiding programmer error in externally evaluating and transcribing constant expressions. They are most often used in declarations. There is no need, however, that constant expressions impose run time costs for their evaluation. They can be evaluated once at compile time or if this is inconvenient because of incompatibilities between the host and object machines, the compiler can generate code for their evaluation at load time. In any case, the resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness and maintainability of programs and does not impose any run time costs.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contributes to ease of learning,

implementing and using a language; allows the user to concentrate on the programming task instead of the language; and leads to more readable, understandable, and predictable programs.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Type transfers hidden in procedure calls with incompatible formal and actual parameters whether intentional or accidental have long been a source of program errors and of programs which are difficult to maintain. On the other hand, there is no reason why the subscript ranges for arrays cannot be passed as part of the arguments. Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

> C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedures execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable, the address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. These are the only two widely used parameter passing mechanisms for data and the many alternatives (at least 10 have been suggested) add complexity and cost to a language without sufficiently increasing the clarity or power. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

> C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters. This generic procedure capability, for example, allows the definition of stacks and queues and their associated operations on data of any given type without knowing the data type when the operations are defined.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation. The use of variable number of argument operations need not and will not cause relaxation of any compile time checks, require use of multiple entry procedures allow the number of actual parameters to vary at run time, nor require special calling mechanisms. If the parameters which can vary are limited to a program specified type treated as any other argument on the call side and as elements of an array within the procedure definition, full type checking can be done at compile time. There will be not prohibition on writing a special case of a procedure for a particular number of arguments.

D. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers
2. Numeric Literals
3. Initial Values of Variables
4. Numeric Range and Step Size
5. Variable Types
6. Pointer Variables

D1. The user will have the ability to associate constant values of any type with identifiers.

The use of identifiers to represent constant values has often made programs more readable, more easily modifiable and less prone to error when the value of a constant is changed. Associating constant values with an identifier is preferable to assigning the value to a variable because it is then clearly marked in the program as a constant, can be automatically checked for unintentional changes, and often can have a more efficient object representation.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types. Regardless of the source of the data and regardless of the object machine the value of constants should be the same. For integers it should be exact and for reals it should be the same within the specified precision. Compiler writers, however, would disagree. They object to this requirement on two grounds: that it is too costly if the host and object machines are different and that it is unnecessary if they are the same. In fact, all costs are at compile time and must be insignificant compared to the life time costs resulting from object cope containing the wrong constant values. As for being unnecessary, there have been all too many cases of different values from program and data literals on the same machine because the compile time and run tima conversion packages were different and imprecise.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

The ability to initialize variables at the time of their allocation will contribute to program clarity, but a requirement to do so would be an arbitrary and sometimes costly decision to the user. Default initial values on the other hand, contribute to neither program clarity nor correctness and can be even more costly at run time. It is usually a programming error if a variable is accessed before it is initialized. It is desirable that the translator give a warning when a path between the declaration and use of a variable omits initialization. Whether a variable will be assigned is in general an unsolvable problem, but it is sometimes determinable whether assignments occur on potential paths. In the case of arrays, it it possible at compile time only to determine that some components (but not necessarily which) have been initialized. There will be provision (at user option) for run time testing for initialization.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications are a special form of assertion. They aid in understanding and determining the correctness of programs. They can also be used as additional information by the compiler in deciding what storage and allocation to use (e.g., half words might be more efficient for integers in the range 0 to 1000). Range specifications also offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds, checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time. Range specifications on approximate valued variables (usually with floating point implemetation) also offer the possibility of their implementation using fixed point hardware.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

There should not be any arbitrary restrictions on the structure of data. This permits arrays to be components of records or arrays and permits records to be components of arrays.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

E. DEFINITION FACILITIES

1. User Definitions Possible
2. Consistent Use of Types
3. No Default Declarations
4. Can Extend Existing Operators
5. Type Definitions
6. Data Defining Mechanisms
7. No Free Union or Subset Types
8. Type Initialization

E1. The user of the language will be able to define new data types and operations within programs.

The number of specialized capabilities needed for a common language is large and diverse. In many cases, there is no consensus as to the form these capabilities should take in a programming language. The operational requirements dictating specific specialized language capabilities are volatile and future needs cannot always be foreseen. No language can make available all the features useful to the broad spectrum of military applications, anticipate future applications and requirements or even provide a universally "best" capability in support of a single application area. *A common language needs capability for growth. It should contain* all the power necessary to satisfy all the applications and the ability to specialize that power to the particular application task. A language with defining facilities for data and operations often makes it possible to add new application-oriented structures and to use new programming techniques and mechanisms through descriptions written entirely within the language. Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions). No programming language can be all things to all people, but a language with data and operation definition facilities can be adapted to meet changing requirements in a variety of areas.

The ability to define data and operations is well within the state of the art. Operation definition facilities in the form of subroutines have been available in all general purpose programming languages since at least the time of early FORTRANs. Data definition facilities have been available in a variety of programming languages for almost 10 years and reached their peak with a large number of extensible languages(Stephen A. Schuman (Ed.) Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, No. 12, December 1971. Also, C. Christensen and C.J. Shaw (Ed.), Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4, August 1969.) (over 30) in 1968 and shortly thereafter. A trend toward more abstract and less machine-oriented data

specification mechanisms has appeared more recently in PASCAL(Niklaus Wirth, "An Assessment of the Programming Language PASCAL, "Proceedings of the International Conference on Reliable Software 21-23 April 1973, p. 23-30). Data type definitions, with operations and data defined together, are used in several languages including SIMULA-67(Jacob Palme, "SIMULA as a Tool for Extensible Program Products, "SIGPLAN Notices, Vol. 9, No. 4, February 1974). On the other hand, there is currently much ferment as to what is the proper function and form of data type definitions.

E2. The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using and implementing the language. If built-in features and user defined data structures and operations are treated in the same way throughout the language so that the base language, standard application libraries and application programs are treated in a uniform manner by the user and by the translator, then these distinctions will grow dim to everyone's advantage. When the language contains all the essential power, when few can tell the difference between the base language and library definitions, and when the introduction of new data types and routines does not impact the compiler and the language standards, then there is little incentive to proliferate languages. Similarly, if typed definitions are processed entirely at compile time and the language allows full program specification of the internal representation, there need be no penalty in run time efficiency for using defined types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

As programmers, we should not expect the translator to write our programs for us (at least in the immediate future). If we somehow know that the translator's default convention is compatible with our needs for the case at hand we should still document the choice so others can understand and maintain our programs. Neither should we be able to delay definitions (possibly forget them) until they cause trouble in the operational system. This is a special case of requirement I1.

E4. The user will be able, within the source language, to extend existing operators to new data types.

When an operation is an abstraction of an existing operation for a new type or is a generalization of an existing operation, it is inconvenient, confusing and misleading to use any but the existing operator symbol or function named. The translator will not assume that commutativity of bulit-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The data objects are given a representation in terms of existing data structures, but they are of little value until operations are available to take advantage of their special properties. When one obtains access to a type, he needs its operations as well as its data. Numeric data is needed in many applications but is of little value to any without arithmetic operations. The definable operations will include constructors, selectors, predicates, and type conversions.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

The above list comprises a currently known set of useful definitional mechanisms for data types which do not require run time support, as do garbage collection and dynamic storage allocation. In conjunction with pointers (see D6), they provide many of the mechanisms necessary to define recursive data structures and efficient sparse data structures.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Free union adds no new power not provided by discriminated union, but does require giving up the security of types in return for programmer freedom. Range and

subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations. Unlike types, membership in subsets can be determined only at run time.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

F. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed
2. Limiting Access Scope
3. Compile Time Scope Determination
4. Libraries Available
5. Library Contents
6. Libraries and Compools Indistinguishable
7. Standard Library Definitions

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access specified in a type definition is necessary to guarantee that changes to data representations and to management routines which purportedly do not affect the calling programs are in fact safe. By rigorously controlling the set of operations applicable to a defined type, the type definition guarantees that no external use of the type can accidentally or intentionally use hidden nonessential properties of the type. Renaming separately defined programming components is necessary to avoid naming conflicts when they are used.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and

costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F3. The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels. The language will use the above lexically embedded scope rules for declarations and other definitions of identifiers to make them easy to recognize and to avoid errors and ambiguities from multiple use of identifiers in a single scope.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

A simple base alone is not sufficient for a common language. Even though in theory such a language provides the necessary power and the capability for specialization to particular applications, the users of the language cannot be expected to develop and support common libraries under individual projects There will be broad support for libraries common to users of well recognized application areas. Application libraries will be developed as early as possible.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

The usefulness of a language derives primarily from the existence and accessibility of specialized application-oriented data and operations. Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. It should be remembered, however, that interfaces cannot be validated at program assembly time without some equivalent of their source language interface specifications, that object modules are machine-dependent and, therefore, not portable, that source code is often more compact than object code, and that compilers for simple languages can sometimes

compile faster than a loader can load from relocatable object programs. Library routines written on other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

Compools have proven very useful in organizing and controlling shared data structures and shared routines. A similar mechanism will be available to manage and control access to related library definitions.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The convenience, ease of use and savings in production and maintenance costs resulting from using high order languages come from being able to use specialized capabilities without building them from scratch. Thus, it is essential that high level capabilities be supplied with the language. The idea is not to provide all the many special cases in the language, but to provide a few general case . hich will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

G. CONTROL STRUCTURES

1. Kinds of Control Structures
2. The Go To
3. Conditional Control
4. Iterative Control
5. Routines
6. Parallel Processing
7. Exception Handling
8. Synchronization and Real Time

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (and no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations and such an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

AD-A037 639 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2

CANDIDATE LANGUAGES EVALUATION REPORT.(U)

JAN 77 B M BROSGOL, R E HARTMAN, J R NESTOR DAHC26-76-C-0006

UNCLASSIFIED IR-217 USACSC-AT-76-11 NL

6 OF 6

ADA037639

END

DATE

FILMED

4-77

The "GO TO" is a machine level capability which is still needed to fill in any gaps which might remain in the choice of structured control primitives, to provide compatibility for translitterating programs written in older languages, and because of the wide familiarity of current practitioners with its use. The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Neither should the language provide specialized facilities which encourage its use in dangerous and confusing ways. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device(Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys, Vol. 6, No. 4, December 1974) provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables t be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. Specialized control structures (e.g., "WHILE DO") have been used for the common situation in which the

termination conditions precedes each iteration. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). A problem which arises in many programming languages is that loop control variables are global to the iterative control and thus, will have a value after loop termination, but that value is usually an accident of the implementation. Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aide program clarity or correctness, and may interfere with the generation of efficient object code. Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

Recursion is desirable in many applications because it contributes directly to their elegance and clairty and simplifies proof procedures. Indirectly, it contributes to the reliability and maintainability of some programs. Recursion is required in order to avoid unnecessarily opaque, complex and confusing programs when operating on recursive data structures. Recursion has not been widely used in DoD software because many programming languages do not provide recursion, practitioners are not familiar with its use, and users fear that its run time costs are to high. Of these, only the run time costs would justify its exclusion from the language.

A major run time cost often attributed to recursion is the need for the presence of a set of "display" registers which are used to keep track of the addresses of the various levels of lexically imbedded environments and which must be managed and updated at run time. The display, however, is necessary only in programs in which routines access variables which are global to their own definition, but local to a more global recursive procedure. This possibility can easily be removed by prohibiting the definition of procedures within the body of a recursive procedure. The utility of such a combination of capabilities is very questionable, and this single restriction will eliminate all added execution costs for nonrecursive procedures in programs which contain recursive procedures.

As with any other facility of the language, routines should be implemented in the most efficient manner consistent with their use and the language should be designed so that efficient implementations are possible. In particular, the most possible regardless of whether the language or even the program contains recursive

procedures. When any routine makes a procedure call as its last operation before exit (and this is quite common for recursive routines) the implementation might use the same data area for both routines, and do a jump to the head of the called procedure thereby saving much of the overhead of a procedure call and eliminating a return. The choice between recursive and nonrecursive routines involves trade-offs. Recursive routines can aid program clarity when operating on recursive data, but can detract from clarity when operating on iterative data. They can increase execution time when procedure call overhead is greater than loop overhead and can decrease execution times when loop overhead is the more expensive. Finally, program storage for recursive routines is often only a small fraction of that for a corresponding iterative procedure, but the data storage requirements are often much larger because of the simultaneous presence of several activations of the same procedure.

> G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

A parallel processing capability is essential in embedded computer applications. Programs must send data to, receive data from, and control many devices which are operating in parallel. Multiprogramming (a form of pseudo paralell processing) is necessary so that many programs within a system can meet their differing real time constraints. The parallel processing capability will minimally provide the ability to define and call parallel processing and the ability to gain exclusive use of system resources in the form of data structures, devices and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. As with recursive routines most accessing and storage management problems can be eliminated by prohibiting complex interactions with other language facilities where the combination has little if any utility. In particular, it will not be possible to define a parallel routine within the

body of a recursive routine and it will not be possible to define any routine including parallel routines within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously. The latter requirement is reasonable for parallel processes because it is information known by the programmer and necessary to the maintainer, because parallel processes cannot normally be stacked, and because it is necessary for the compilation of efficient programs.

G7. The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

It is essential in many aplications that there be no program halts beyond the user's control. The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure. The latter form requires an exception handling formal parameter class (see C7).

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program.

These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

H. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics
2. No Syntax Extensions
3. Source Character Set
4. Identifiers and Literals
5. Lexical Units and Lines
6. Key Words
7. Comment Conventions
8. Unmatched Parentheses
9. Uniform Referent Notation
10. Consistency of Meaning

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

Clarity and readability of programs will be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers and conventional forms allows the programmer to use notations which have their familiar meanings to put down his ideas and intentions in the order and form that humans think about them, and to transfer skills he already has to the solution of the problem at hand. A simple uniform language reduces the number of cases which must be dealt with by anyone using the language. If programs are difficult for the translator to parse they will be difficult for people. Similar things should use the same notations with the special case processing reserved for the translator and object machine. The purpose of mnemonic identifiers and key words is to be informative and increase the distance between lexical units of programs. This does not prevent the use of short identifiers and short key words.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

If the user can change the syntax of the language, then he can change the basic character and understanding of the language. The distinction between

semantic extensions and syntactic extensions is similar to that between being able to coin new words in English or being able to move to another natural language. Coining words requires learning those new meanings before they can be used, but at the same time increases the power the language for some application areas. Changing the grammar, (e.g., Franglais, the use of French grammar with interspersed English words) however, undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable, but there should be no provision for changing the grammatical rules of the language. This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

A common language should use notations and a character set convenient for communicating algorithms, programs, and programming techniques among its users. On the other hand, the language should not require special equipment (e.g., card readers and printers) for its use. The use of the 64 character ASCII subset will make the language compatible with the federal information processing standard 64 character set, FIPS-1, which has been adopted by the U.S.A. Standard code for Information Interchange (USASCII). The language definition will specify the translation from the publication language into the restricted character set.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Lexical units of the language should be defined in a simple uniform and easily understood manner. Some possible break characters are the space(W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Moore, Academic Press, 1972. & Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). A literal break character contributes to the readability of

programs and makes the entry of long literals less error prone. With a space as a break character one can enter multipart (i.e., more than one lexical unit) identifiers such as "REAL TIME CLOCK" or long literals, such as, "3.14159 26535 89793." Use of a break can also be used to guarantee that missing quote brackets on character literals do not cause errors which propagate beyond the net end-of-line. the language should require separate quoting of each line of a long literal: "This is a long" "literal string".

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

Many elementary input errors arise at the end of lines. Programs are input on line oriented media but the concept of end-of-line is foreign to free format text. Most of the error prone aspects of end-of-line can be eliminated by not allowing lexical units to continue over lines. The sometimes undesirable effects of this restriction can be avoided by permitting identifiers and literals to be composed from more than one lexical unit (see H4) and by evaluating constant expressions at compile time (see C4).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations or the are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Key words will be few in number because each new key word introduces another case in the parsing rules and thereby adds to complexity in understanding the language, and because large numbers of key words inconvenience and complicate the programmer's task of choosing informative identifiers. Key words should be concise, but being information is more important than being short. A major exception is the key word introducing a comment; it is the comment and not its key word which should do the informing. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

These are all obvious points which will encourage the use of comments in programs and avoid their error prone features in some existing languages. Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H8. The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. This makes programs easier to write because it sometimes saves writing several "ENDs" at the end of programs and because it eliminates all syntax errors for missing "ENDs." Failure to require proper parentheses matching makes it more difficult to write correct programs. Good programming practice requires that matching parentheses be included in programs whether or not they are required by the language. Unfortunately, if they are not required by the language then there can be no syntax check to discover where errors were made. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2...sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H9. There will be a uniform referent notation.

The distinction between function calls and data reference is one of representation, not of use. Thus, there will be no language imposed syntactic distinction between function calls and data selection. If, for example, a computed function is replaced by a lookup table there should be no need to change the calling program. This does not preclude the inclusion of more than one referent notation.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

This contributes to the clarity and uniformity of programs, protects against psychological ambiguity and avoids some error prone features of extant languages. In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/I subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function calls). It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic
2. Object Representation Specifications Optional
3. Compile Time Variables
4. Conditional Compilation
5. Simple Base Language
6. Translator Restrictions
7. Object Machine Restrictions

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

The language should be oriented to provide a high degree of management control and visibility to programs and toward self-documenting programs with the programmer required to make his decisions explicit. On the other hand, the programmer should not be forced to overspecify his programs and thereby cloud their logic, unnecessarily eliminate opportunities for optimization, and misrepresent arbitrary choices as essential to the program logic. Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

The capabilities available in any language can be partitioned into two groups, those which are definable within the base and those which provide an essential primitive capability of the language. The smaller and simpler the base the easier the language will be to learn and use. A clearly delineated base with features not in the base defined in terms of the base, will improve the ease and efficiency of learning, implementing and maintaining the language. Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating speciaized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. In each case, however: (a) some limit must be set, (b) whatever the limit, it will impose on some and therefore must be known by the users of the translator, (c) letting each translator set its own limits means that programs will not be portable, (d) setting the limits very high requires that the translator be hosted only on large machines and (e) quite low limits do not impose significantly on either the power of the language or the readability of programs. Thus, the limits should be set as part of the language definition. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code
2. Optimizations Do Not Change Program Effect
3. Machine Language Insertions
4. Object Representation Specifications
5. Open and Closed Routine Calls

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The base language and library definitions might contain features and capabilities which are not needed by everyone, or at least, not be everyone all the time. The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language . The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main store and backing shore which is not under program control (unless the object machine has virtual memory with underlying management beyond the control of all its users). Language features will result in special efficient object codes when their full generality is not used. A large number of special cases should compile efficiently. For example, a program doing numerical calculations on unsubscripted real variables should produce code no worse than FORTRAN. Parameter passing for single argument routines might be implemented much less expensively than multiple argument routines.

One way to reduce costs for unneeded capabilities is to have a base language whose data structures and operations provide a single capability which is composable and has a straight-forward implementation in the object code of conventional architecture machines. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J2. Any optimizations performed by the translator will not change the effect of the program.

More simply, the translator cannot give up program reliability and correctness, regardless of the excuse. Note that for most programming languages there are few known safe optimizations and many unsafe ones. The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

It is difficult to be enthusiastic about machine language insertions. They defeat the purpose of machine independence constrain the implementation techniques complicate the diagnostics, impair the safety of type checking, and detract from the reliability, readability, and modifiability of programs. The use of machine language insertions is particularly dangerous in multiprogramming applications because they impair the ability to exclude, "a priori," a large class of time-dependent bugs. Rigid enforcement of scope rules by the compiler in real time applications is a powerful tool to ensure that one sequential process will not interfere with others in an uncontrolled fashion. Similarly, when several independent programs are executed in an interleaved fashion, the correct execution of each may depend on the others not having improperly used machine language insertions.

Unfortunately machine language insertions are necessary for interfacing special purpose devices, for accessing special purpose hardware capabilities, and for certain code optimizations on time critical paths. Here we have an example of Dijkstra's dilemma in which the mismatch between high level language programming and the underlying hardware is unacceptable and there is no feasible way to reject the hardware. The only remaining alternative is to "continue bit pushing in the old way, with all the known ill effects." Those ill effects can, however, be constrained to the smallest possible perimeter in practice if not in theory. The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile

time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It is often necessary to give detailed specifications of the object data representations to obtain maximum density for large data files to meet format requirements imposed by the hardware of peripheral equipment, to allow special optimizations on time critical paths, or to ensure compatibility when transferring data between machines.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3). When multiple fields per word are specified the compiler may have to generate some form of shift and mask operations for source program references and assignments to those variables (i.e., fields). As with machine-language insertions, object data specifications should be used sparingly and the language features for their use must be Spartan, nor grandiose.

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator. The translator might, for example, attempt to minimize access time and/or memory space in determining the object representation. It might, depending on the object machine characteristics, assign variables and fields of records to full words, but assign array elements to the smallest of bits, bytes, half words, words or exact multiple words permitted by the logical description.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

The use of inline open procedures can reduce the run time execution costs significantly in some cases. There are the obvious advantages in eliminating the parameter passing, in avoiding the saving of return marks, and in not having to pass arguments to and from the routine. A less obvious, but often more important advantage in saving run time costs is the ability to execute constant portions of routines at compile time and, thereby, eliminate time and space for those portions of the procedure body at run time. Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.

K. PROGRAM ENVIRONMENT

1. Operating System Not Required
2. Program Assembly
3. Software Development Tools
4. Translator Options
5. Assertions and Other Optional Specifications

K1. The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

A language definition cannot dictate the architecture of existing object machines whether defined entirely in hardware or in a hardware/software combination. It can provide a source language representation of all the needed capabilities and attempt to choose these so they have an obvious and efficient translation in the object machines.

K2. The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4). In general, separate compilation increases the difficulty and expense of the interface validations needed for program safety and reliability and detracts from object program efficiency by removing many of the optimizations otherwise possible at the interfaces, but at the same time it reduces the cost and complexity of compilation.

K3. A family of programming tools and aids in the form of support packages including linkers, loaders and debugging systems will be made available with the language and its translators. There will be a consistent easily used user interface for these tools.

The time has passed in which a programming language can be considered in isolation from its programming environment. The availability of programming tools which need not be developed and/or supported by individual projects is a major factor in the acceptability of a language. There is no need to restrict the kinds or form of support software available in the programming environment, and continued development of new tools should be encouraged and made available in a competitive market. It is, however, desirable that tools be developed in their own source language to simplify their portability and maintainability.

K4. A variety of useful options to aid generation, test, documentation and modification of programs will be provided as support software available with the language or as translator options. As a minimum these will include program editing, post- mortem analysis and diagnostics, program reformating for standard indentations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation and modification of programs. The "best" set of capabilities and their proper form is not currently known. Since nonstandard translator options and availability of nonstandard software tools and aids do not adversely affect software commonality, the language definition and standards will not dictate arbitrary choices. Instead, the development of language associated tools and aids will be encouraged within the constraint of implementing and supporting the source language as defined. Tools and debugging aids will be source language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is false. It might provide run time tracing of specified program variables. Dimensional analysis might be done on units of measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run time statistics. There might be translator supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs and/or an estimate of their execution times. It might provide a variety of listing options.

K5. The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications;, and units of measures in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

There are many opinions on the desirability, usefulness, and proper form for each of these specifications. Better program documentation is needed and specifications of these kinds may help. Specifications also introduce the possibility of automated testing, run time verification of predicates, formal program proofs, and dimensional analysis. The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions and units of measure in source language programs should be enclosed in special brackets and should be treated as interpreted comments -- comments which are delimited by special comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc.--but whose interpretation would be optional to translator implementations.

L. TRANSLATORS

1. No Superset Implementations
2. No Subset Implementations
3. Low-Cost Translation
4. Many Object Machines
5. Self-Hosting Not Required
6. Translator Checking Required
7. Diagnostic Messages
8. Translator Internal Structure
9. Self-Implementable Language

L1. No implementation of the language will contain source language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This guarantees that use of programs and software subsystems will not be restricted to a particular site by virtue of using their unique version of the language. It also represents a commitment to freezing the source language, inhibiting innovations and growth in the form of the source language, and confining the base language to the current state of the art in return for stability, wider applicability of software tools, reusable software, greater software visibility, and increased payoff for tool building efforts. It does not, however, disallow library definition optimizations which are translator unique.

L2. Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

If individual compilers implement only a subset of the language, the there is no chance for software commonality. If a translator does not implement the entire language, it cannot give its users access to standard supported libraries or to application programs implemented on some other translator. Requiring that the full language be implemented will be expensive only if the base language is large, complex, and nonuniform. The intended source language product from this effort is a small simple uniform base language with the specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple low cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L3. The translator will minimize compile time costs. A goal of any translator for the language will be low cost translation.

Where practical and beneficial the user will have control over the level of optimization applied to his programs. The programmer will have control over the tradeoffs between compile time and run time costs. The desire for small efficient translators which can be hosted by machines with limited size and capability should influence the design of the base language against inclusion of unnecessary features and towards systematic treatment of features which are included. The goal will be effective use of the available machines both in object execution and translation and not maximal speed of translation.

Translation costs depend not only on the compiler but the language design. Both the translator and the language design will emphasize low cost translation, but in an environment of large and long-lived software products this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations. This means that the inherent costs of specific language features is the context of the total language must be understood by the designers, implementers and users of the language. One consequence should be that trivial programs compile and run in trivial time. On the other hand, significant optimization is not expected from a minimal cost translator.

L4. Translators will be able to produce code for a variety of object machines. The machine independent parts of translators might be built independent of the code generators.

There is currently no common widely used computer in the DoD. There are at least 250 different models of commercial machines in use in DoD with many more specialized machines. A common language must be applicable to a wide variety of models and sizes of machines. Translators might be written so they can produce object code for several machines. This reduces the proliferation of translators and makes the full power of an existing translator available at the cost of producing an additional code generator.

L5. The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The DoD operational programming environment includes many small machines which are unable to support adequately the design, documentation, test, and

debugging aids necessary for the development of timely, reliable or efficient software. Large machine users should not be penalized for the restrictions of small machines when a common language is used. On the other hand, the size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

> L6. The translator will do full syntax checking, will check all operations and parameters for type compatibility and will verify that all language imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

The purpose of source language redundancy and avoidance of error prone language features is reliability. The price is paid in programmer inconvenience in having to specify his intent in greater detail. The payoff comes when the translator checks that the source program is internally consistent and adheres to its authors' stated intentions. There is a clear trade-off between error avoidance and programming ease; surveys conducted in the Services show that the programmers as well as managers will opt for error avoidance over ease when given the choice. The same choice is dictated by the need for well documented maintainable software.

> L7. The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

The translator will attempt to provide the maximal useful feedback to its user. Diagnostic messages will not be coded but will be explanatory and in source language terms. Translators will continue processing and checking after errors have been found but should be careful not to generate erroneous messages because of translator confusion. The translator will always produce correct code; when source programs errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause a run time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages provided as part of the language definition contributes to commonality in the implementation and use of the language. The discipline of designing diagnostic messages keyed to the design may also uncover pitfalls in the language design and thereby contribute to a more precise and better understood language description.

L8. The characteristics of translator implementations will not be dictated by the language definition or standards.

The adoption of a common language is a commitment to the current state of the art for programming language design for some duration. It does not, however, prevent access to new software and hardware technology, new techniques and new management strategies which do not impact the source language definition. In particular, innovation should be encouraged in the development of translators for a common language providing they implement exactly the source language as defined. Translators like all computer programs should be written in expectation of change.

L9. Translators for the language will be written in their own source language.

There will be at least one implementation of the translator in its own language which does all parsing and compile-time checking and produces an output suitable for easy translation to specific object machines. If the language is well-defined and uniform in structure, a self-description will contribute to understanding of the language. The availability of the machine independent portion of a translator will make the full power of the language available to any object machine at the cost of producing an additional code generator (whose cost may be high) and it reduces the likelihood of incompatible implementations. Translators written in their own source language are automatically available on any of their object machines providing the object machine has sufficient resources to support a compiler.

M. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only
2. Unambiguous Definition
3. Language Documentation Required
4. Control Agent Required
5. Support Agent Required
6. Library Standards and Support Required

M1. The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and is compatible with "best" current programming practice but the design and implementation of the language should not require additional research or require use of untried ideas. State-of-the-art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features and simplifies the whole.

M2. The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

A complete and unambiguous definition of a common language is essential. Otherwise each translator will resolve the ambiguities and fill in the gaps in its own

unique way. There are currently a variety of methods for formal specification of programming language semantics but it remains a major effort to produce a rigorous formal description and the resulting products are of questionable practical value. The real value in attempting a formal definition is that it reveals incomplete and ambiguous specifications. An attempt will be made to provide a formal definition of any language selected but success in that effort should not be requisite to its selection. Formal specification of the language might take the form of an axiomatic definition, use of the Vienna Definition Language, or use of some other formal semantic system.

> M3. The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine level language of an abstract digital computer.

The language should be intuitively correct and easily learned and understood by its potential users. The language definition might include an Algol-60 like description(P. Naur (Ed.), "Revised Report on the Algorithmic Language Algol-60," Communication of the A.C.M. Vol.6, No. 1, January 1963, p. 1-17.) with the source language syntax given in BNF or some other easily understood metalanguage and the corresponding semantics given in English. As with the descriptions of digital computer hardware, the semantics and syntax of each feature must be defined precisely and unambiguously. The action of any legal program will be determinable from the program and the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities which are built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics and examples of each language construct, listings of all key words and language defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

> M4. The language will be configuration managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

Without controls a hopefully common language may become another umbrella under which new languages proliferate while retaining the same name. All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M5. There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids.

Language commonality is an essential step in achieving software commonality, but the real benefits accrue when projects and contractors can draw on existing software with assurance that it will be supported, when systems can build from off the shelf components or at least with common tools, and when efforts can be spent to expand existing capabilities rather than building from scratch. Support of common widely used tools and aids should be provided independent of projects if common software is to be widely used. Support should be on a DoD-wide basis with final responsibility resting with a stable group or groups of qualified in-house personnel.

M6. There will be standards and support agents for common libraries including application-oriented libraries.

In a given application of a programming language three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions but not for the language and its libraries which are used by many projects and sites. A principal user might act as agent for an entire application area.

Appendix III

REVIEW OF "TINMAN"

Section I. INTRODUCTION

1. Overall Reaction.

At the outset, it must be emphasized that the requirements described by the "Tinman" are believed to be an excellent overall measure by which a candidate Common DoD Language can be assessed. Should the DoD obtain a language which fits these requirements closely, the quality of DoD software can be expected to increase significantly per unit cost. The comments given in this chapter are made with the objective of allowing the requirements of the Tinman to be further enhanced. While, of necessity, these comments suggest changes, they should be interpreted in the positive context stated above.

2. Notational Conventions Used in This Chapter.

The comments on the Tinman characteristics are classified as follows:

a. Ambiguity. The requirements are not clear in these areas. Several interpretations are possible.

b. Inconsistency. Two or more requirements are in conflict.

c. Overspecification. Although the requirements specify a reasonable feature, there are other features which seem to satisfy the intent of the requirement yet are ruled out by the specific details of the requirement.

d. Disagreement. These are requirements which we feel are not appropriate.

e. Comment. Additional comments which do not fit any of the above are included here.

Section II. DATA AND TYPES

1. Typed Language (A1).

a. Overspecification, Ambiguity.

(1) The Tinman states that "the language will be typed" and that all types will be "determinable at compile time and unalterable at run-time". Types are considered properties of variables, expressions, etc. However, according to the Tinman, not all properties are included in the type. For example, according to A3, precision of reals does not distinguish type. According to A6, the upper bound for array subscripts can be determinable at run-time and is therefore not a property of an array type. It may be noted that two arrays which differ only in their upper bound will have the same type, yet one cannot be assigned to the other.

(2) The ambiguity of these requirements is due to not carefully classifying those properties which distinguish types, and not specifying the effect of those properties which do not distinguish types.

(3) The overspecification of requirements here is due to the partition of those properties which distinguish type from those that don't. This separation is somewhat arbitrary and a different partition does not necessarily result in languages with any differences in their behavior. The difference is strictly descriptive. A language that chooses a different partition than that reflected by the Tinman requirements will be in conflict with these requirements even when it completely satisfies the intent of the requirements.

(4) We recommend that the requirements be changed as follows. First, rather than describe the requirements in terms of types, describe them in terms of properties of variables, expressions, etc. Second, the requirement should classify the properties by whether or not they must be known at

compile time and by what effect they have on behavior. Some properties (e.g., packing of representation) don't affect the value set at all. Other properties (e.g., precision, range, etc.) affect value sets but not the set of applicable operations. Still other properties (e.g., upper bound of arrays) do affect the set of valid operations. Finally, the requirement should be changed so that it discusses which classes of properties are permitted to be different in various operations (e.g., assignment, parameter passing, etc.).

b. Ambiguity, Inconsistency. All types "will be determinable at compile time". This requirement might be considered to be in conflict with C8, "specification of type will be optional in procedure declarations".

2. Data Types (A2).

Disagreement. The Tinman alleges that the listed types and type generators are sufficient "to efficiently mechanize any other desired type". It seems likely that realizations of pointers and "monitored variables", for example, would probably be quite difficult to achieve by these means. Consequently, we would include a typed-pointer type in the given list, and perhaps soften the stated claim a bit.

3. Precision (A3).

Disagreement. "Precision specification will apply to arithmetic operations." Although this may be acceptable for local variables with the same default precision as the operations, operations whose operands are non-local variables with a different default precision will use the wrong precision. It can be argued that most arithmetic with reals is likely to be performed with "single precision" values, and that the FORTRAN style of specifying only exceptions to this rule would suffice. Yet, the question must be raised as to whether the general trend towards explicit expression of intent ought not be adhered to in this context: namely, that each variable declaration should include specifications of its range and precision of values. For efficiency reasons, user-stated requirements of

precision will be adjusted upwards to correspond to hardware-provided single and double precision forms. Evaluation of operations, as in virtually all languages which recognize multiple precisions, will be conducted at the largest of the precisions of the operands. Requiring "a single specification of the precision" will cause programmers to specify the largest precision needed for any object in the scope. It is not clear that this contributes to readability, implementability, or quality of the software.

4. Fixed Point Numbers (A4).

a. Disagreement. "Fixed point numbers will be treated as exact quantities." For a given step size, not all operations will in general yield results with the same step size. For example, x := x/2 will be illegal. We recommend that fixed point numbers, like floating-point numbers, be treated as approximate quantities.

b. Disagreement. Two prominent reasons inflate the cost of software using fixed-point rather than floating-point arithmetic. First, the scaling rules, when automatically provided, are intricate, and difficult to use without error. Second, and more important, programmers must dedicate their attention to a consideration which is absent from floating-point calculations: the precision and scaling of intermediate values formed during the evaluation of expressions. Surprising losses of important bits of value can occur in these cases, which can be very difficult to locate; reliability thus drops, and debugging costs soar. We believe that automatic scaling of fixed-point values both encourages the use of fixed-point (which should be discouraged), and causes the introduction of subtle bugs through implicit actions invisible to the reader of the program. Consequently, automatic scaling should be minimized.

5. Character Data (A5).

Ambiguity. This requirement might be interpreted to mean that the lexical analyzer of a compiler must be extensible. This would introduce undesired complexity. This requirement should be clarified.

6. Arrays (A6).

a. Overspecification, Disagreement. "The number of dimensions, the [element] type, and the lower subscript bound [for the array] should be determinable at compile time." The specified distinction between upper and lower subscript bounds appears to offer idiosyncrasy without compensating advantage. We suggest that it would be better either 1) to require all integer array subscripts to begin at the same value (e.g., 1), or 2) to make the rules for determination of upper and lower bounds be identical, or 3) to write requirements so either of the solutions 1 or 2 is permitted.

b. Overspecification. This requirement implies that multidimensional arrays will be required. A similar effect can be achieved by composing multiple single dimensional arrays. We recommend that the requirements be changed to permit this alternate approach.

7. Records (A7).

a. Overspecification. This requirement specifies that overlaid data will be achieved via the record data structure. Some languages (e.g., CS-4) satisfy the intent but not the details of this requirement by having a separate discriminated union data structure.

b. Overspecification. The requirement states that the discrimination condition can be "any expression". We believe that restricted discrimination conditions (e.g., discrimination on the name of each overlay) do not significantly limit the power of the language. They may even result in more readable programs since the selected overlay is more apparent.

c. Comment. It should be noted here that Pascal provides such a discriminated union capability but with a mechanism that allows users to defeat type-rule enforcement as thoroughly as Fortran EQUIVALENCE does. To ensure the desired degree of safety, the desired degree of safety should be made part of the requirement.

Section III. OPERATORS

1. Assignment and Reference (B1).

Disagreement. "Assignment and reference operations should be automatically defined for all data types." Assignment and reference, like other operations on data objects, should be subject to access regulation. The implementor of a "semaphore" data type, for example, will not wish any assignments to occur other than by means of "primitive" operations he makes available to semaphore users.

2. Equivalence (B2).

a. Ambiguity. Comparison of any two data objects. Four different terms or phrases are used in this section to describe the same comparison, resulting in serious ambiguity: "identity", "equivalence", "equality", and "multiple references to the same abstract data object".

b. Ambiguity. The text leaves a question as to whether it is intended that comparisons should be valid between objects of different types; the strong implication is that such comparisons would be permitted. We believe that a compile-time error message should result rather than a FALSE value, since the expression is likely to be a mistake.

c. Disagreement. "For floating-point numbers, identity will be defined as the same within the specified minimum precision." It is important to distinguish between the declared precision of a real variable and its effective precision as the result of a computation. In general the effective precision will be less than the declared precision. It would be beneficial for the requirements to specify a need for comparison operations which accept an additional argument specifying the precision to which the comparison is to be performed.

3. Relationals (B3).

a. Ambiguity. "Numbers and types defined by enumeration have an obvious ordering..." In an expression-oriented language, which is not precluded by the Tinman, there are contexts in which the enumeration order of a type is not obvious. If there is a canonical ordering for enumerated

types, this problem is resolved, but at the expense of eliminating order determined by the sequence in which values are written by the user in type definitions.

b. Overspecification. For user-defined types, the ordering of the resulting abstract type is not always the same as that of the representational type. We believe that the requirement should be that it is possible to define either ordered or unordered types. The inclusion of an automatic ordering together with an override constitutes an overspecification.

4. Arithmetic Operations (B4).

Overspecification. The requirements state that division will have a real result. This requirement is due to the fact that the integers are not closed under division. The Tinman-proposed answer for this problem is not the only reasonable solution, however. Equally reasonable alternatives are (a) to define integer division to produce an integer result by truncation, (b) to define integer division to be legal only for those operand values which produce an integer result (i.e., with a zero remainder), and (c) to disallow integer division entirely.

5. Boolean Operations (B6).

Inconsistency. "The operations and and or on scalars should be evaluated in short circuit mode." We agree with the requirement, as far as it goes. However, in accordance with E2, the capability for defining such operations should be made available to users. If and and or are built in, as implied, and if formal parameter binding classes are as restricted as in C7, the user does not have this capability.

6. Scalar Operations (B7).

a. Ambiguity. "The source language should permit scalar operations and assignment on conformable arrays and should permit data transfers between records or arrays of identical structure." There is a terminology change here whose intention is unclear: "data transfers" vs. "assignment and reference operations" (B1), and "structure" vs. "type" (also B1).

b. Ambiguity, Disagreement. The wording of the section clearly implies that it should be possible to add a (3 x 4) array of integers to a (2 x 3 x 2) array of integers. This requirement should stipulate instead that the arrays be of the same rank and that corresponding dimension sizes be equal.

7. Type Conversion (B8).

a. Ambiguity. It is not clear from this requirement whether mixed type operations (e.g., adding an integer to a real) are to be permitted. Mixed type operations can be considered to be either (a) operations which involve implicit conversions (prohibited by B8), or (b) different instances of a generic operation (permitted by C8).

b. Disagreement. "No conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter." There seems to be no strong reason for allowing this exception to the general rule against implicit type conversions. This exception can be a problem for parameters which are bound to variables (see C7). In this case the value of the formal parameter "constituent" can be changed, but not the particular constituent. This results in complex rules for assignment to the formal.

8. Changes in Numeric Representation (B9).

Inconsistency, Disagreement. Range checking is "optional for hardware installations which do not have overflow detection". This is in conflict with requirements B1 and B2 of Section V, which require that all implementations be the same. Even when hardware checking is not possible, software checks can always be performed. Since these software checks may have a severe impact on efficiency, their presence should be under the user's control (rather than under the implementer's control).

9. I/O Operations (B10).

Comment. We agree with the statements in B10, yet feel obliged to point out the troublesome lifetime issues raised by the need for these capabilities. Because the lifetime of data on files can exceed the lifetime of the processes or even programs in which the relevant type definitions occur,

and because files may be transferred to different installations (where different type definitions are in effect), there is a significant difficulty in performing rigorous type checking on filed data. Relaxation or elimination of type checking on files can lead to undetectable errors and to use of files as a means to defeat type checking.

10. Power Set Operations (B11).

Overspecification. The effect of a power set type can be achieved by permitting array indices to be enumeration types. An array of Booleans indexed by an enumeration type is quite similar to a power set. We recommend that this requirement be changed to include this alternate approach.

Section IV. EXPRESSIONS AND PARAMETERS

1. Side Effects (C1).

<u>Ambiguity, Comment</u>. "Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left to right." The ambiguity here is that it is not stated what kinds of things are considered to be side effects. There are several possibilities, including (a) changes to variables that are (or could be) referenced in other arguments, (b) performing I/O, and (c) signalling exception conditions. If all of these (particularly c) are considered side effects, then the freedom of an optimizer to reorder argument evaluation is severely restricted. Possibly exception conditions should not be so severely restricted with respect to evaluation order.

2. Operand Structure (C2).

a. <u>Comment</u>. "Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader." This probably is possible only if explicit parentheses are required everywhere. Even the APL approach, with no operator hierarchy, fails to satisfy the "obvious"-requirement with every reader.

b. <u>Ambiguity, Disagreement</u>. "The user should not be able to define new operator precedence rules nor change the precedence of existing operators." We agree with this statement as worded; however, its intention is apparently to prohibit the definition of new operator precedence <u>levels</u> (not <u>rules</u>). We take strong exception to this intention. If operator hierarchy is supported at all (* being stronger than +, for example), it seems simply a caprice to prohibit all future designers of application packages from implementing operator sub-hierarchies meaningful to both their application area and the existing hierarchy for integers, Booleans, etc. If any restrictions of Tinman type are to be imposed, let them be uniformly applied to the "built-in" types as well, not just at the boundary between what we now know well (arithmetic and logical operations on built-in types) and what we have yet to exploit (user-defined types and operations). This boundary will change with experience: the language rules should not obstruct this experience.

3. Constant Expressions (C4).

<u>Comment</u>. "Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time." For the term "constant expression" to be meaningful, the language must clearly specify the rules for determining when an expression is "compile-time (or load-time) evaluable". The host/target machine differences here can be quite troublesome, requiring, in the limit, a target-machine simulator running on the host machine as part of the compiler. We recommend a cautious approach in this area so as not to burden "small" implementations with the necessity for elaborate pre-run-time evaluations of program segments.

4. Consistent Parameter Rules (C5).

a. <u>Ambiguity</u>. This requirement implies the existence of parameters to declarations. The meaning of this concept is obscure.

b. <u>Inconsistency</u>. "Uniformity and consistency contributes to ease of learning, implementing and using a language." This requirement together with requirement B6 which requires short circuit evaluation of certain Boolean operations, is in conflict with C7 which provides no parameter mechanism for user-defined short-circuited behavior.

6. Type Agreement in Parameters (C6).

<u>Disagreement</u>. "Formal parameters of a union type should be considered conformable to actual parameters of any of the component types." See the comment given for B8, above.

7. Formal Parameter Kinds (C7).

a. <u>Disagreement</u>. We strongly disagree with the oversimplified view of the formal parameter mechanism, one of the most key issues in modern programming language design for reliable software. We recommend that three classes of binding be provided for data: COPY, REF, and NAME. COPY and NAME correspond to classes supported in Algol 60, while REF is essentially from Fortran. Each should be viewed in a more general way than in their original environment: the intention of the procedure should be further detailed for

program readability and efficiency reasons. By specifying whether the procedure wishes to obtain the value of the actual parameter, modify the actual parameter, or both, the user enables the compiler to perform exactly appropriate validity checking on the correspondence between actual and formal parameters, and to avoid generation of code for transmission of data to or from the procedure when such transmission is not to be performed. While a skeptic can claim that this produces nine binding classes, such a claim obfuscates the fact that two <u>separate</u> properties are each being specified as one of three possibilities. This seems far superior to requiring that a large array be vulnerable to accidental alteration because it was received as a "read-write" parameter to avoid the space and time costs of making a local copy; or, to insisting that an integer be subjected to indirect addressing for many, many accesses by a procedure because its final value is to be returned as an output. NAME binding is exceptionally efficient when used with open expansion of procedure bodies, especially the type of open expansions which are necessary in the implementation of data abstractions. The reconsideration of the Tinman views in this area is urgently recommended.

b. <u>Overspecification</u>. "In addition there will be a formal parameter class for specifying the control actions when exception conditions occur...." This is only one of several possible ways of handling exception conditions. The requirements should be loosened to permit other equally attractive exception-handling mechanisms.

c. <u>Disagreement</u>, <u>Overspecification</u>. "In addition there will be a formal parameter class ... for procedure parameters." There is no strong reason why procedural parameters cannot be passed using one of the standard binding classes. We also disagree with this requirement. There seems to be no strong need for this facility which introduces considerable complexity to both the language and its implementation.

8. Formal Parameter Specifications (C8).

a. <u>Ambiguity</u>, <u>Disagreement</u>. "Specification of the type, range, precision, scale, and format of parameters will be optional." "...permits the writing of generic procedures...." "...eliminates the need for compile-time <u>type</u> parameters." The described capability seems to be that

of a substitution text-macro, which can do what text macros can do and no more. This falls far short of what is needed to properly implement generic procedures. Compile-time parameters and enquiries are essential.

b. Inconsistency. This requirement is inconsistent with A1, which requires that all types be known at compile time. See our comments for A1, above.

c. Ambiguity. The meaning of "generic" procedures is quite vague. There are at least two possible interpretations. First, the requirement might refer to a true "generic" procedure--a procedure that has several bodies, one of which is selected for each call based upon the types of the actual parameters. Another interpretation is that the requirement refers to a "polymorphic" procedure--a procedure that has a single body that uses generic operations. The meaning of a specific call is given by selecting the appropriate body for each of the referenced generic operations. (For example, a not-equal polymorphic operation can be implemented in terms of a Boolean not operation and a generic equal operation.) Both polymorphic and generic operations seem to be desirable features.

9. Variable Number of Parameters (C9).

a. Comment. "There should be provision for variable numbers of arguments, but in such cases all but a constant number of them should be of the same type." "There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as print, ..." The described capability will be useful in implementing a function such as print only if the language restricts the members of its argument list to all being of the same type. Not even Fortran imposes such a confining restriction.

b. Disagreement. The advantages offered by permitting a variable number of arguments is more than offset by the added complexity introduced into the language. We recommend that this requirement be dropped.

Section V. VARIABLES, LITERALS, AND CONSTANTS

1. Numeric Literals (D2).

<u>Comment</u>. "Literals are needed for all atomic data types...." Literals are also desirable for character strings. In fact, to complete the data type definition facilities, it would be meaningful to provide a method for prescribing literals of defined types. However, no known language has this ability, and we have no specific recommendation in this area at the moment. The "constructor" function notion is adequate in the meantime.

2. Numeric Range and Step Size (D4).

a. <u>Ambiguity</u>. "Range and step size specifications will not be interpreted as defining new types." See the comments under A1.

b. <u>Inconsistency</u>. "...Range specifications can not in general be evaluated at compile time." This requirement together with variable parameters (reference binding) (see C7) can have a serious impact on the efficiency requirements of J1.

3. Variable Types (D5).

a. <u>Ambiguity, Disagreement</u>. "The range of values... will be any built-in type." This implies that types are sets of values. We strongly disagree. See our comments under A1.

b. <u>Inconsistency</u>. "The range of values will be... a contiguous subsequence of any enumeration type." This is in conflict with E7 which states "Type definition by... subsetting [is] not desired".

4. Pointer Variables (D6).

a. <u>Overspecification, Disagreement</u>. "Consequently pointer/nonpointer will be a property only of variables for composite types and of composite array and record components." While the Tinman's desire for safety and control is correctly motivated, a more uniform approach, having the desired safety, is also possible. Pointer-type definitions can be required to designate the one type of

object to which they may point; in this way, no type defeating is possible, as it is with "absolute address" types. A heap storage class should also be supplied, allowing explicit allocation and de-allocation of objects to which pointers may point. Discriminated unions allow orderly realization of pointers which can point to objects of more than one type. No computational operations on pointers need be provided, ensuring that incorrect values are never generated.

b. Disagreement. "The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable." In dynamically modified list structures the lifetimes of individual "nodes" are such that this requirement cannot be met. Possible solutions include a heap garbage collector or a system of run-time checks. The garbage collector costs in time and space are frequently quite undesirable or intolerable in real-time applications. Run-time checks can be thoroughly effective, but there appear to be significant problems in combining programs which include such run-time checks with program segments (such as libraries) when the latter have been thoroughly debugged and the run-time checks removed. These problems arise only when checked and unchecked segments communicate via pointers, which is a relatively infrequent case.

Section VI. DEFINITION FACILITIES

1. User Definitions Possible (E1).

<u>Ambiguity</u>. "The user of a language will be able to define new data types...." This requirement is very vague. What is a data type? (See our comments under A1.) Are new data types restricted to simply compositions of existing types and type generators, or is entirely new behavior possible? Do user-defined types have the same "power" as built-in types? Are representations to be hidden?

2. Consistent Use of Types (E2).

<u>Ambiguity</u>, <u>Disagreement</u>. "The 'use' of defined types will be indistinguishable from [that of] built-in types." The intent of this requirement is not clear. If it implies that a user-defined type can "simulate" the behavior of any built-in type, then realizing this requirement will be extremely difficult if not impossible with current technology.

3. Data Defining Mechanisms (E6).

<u>Ambiguity</u>, <u>Disagreement</u>. This seems to be a restriction on the types on which a newly-defined type can be based. It is overly restrictive. For example, it seems reasonable to permit a new user-defined type to depend upon a previously user-defined type. We recommend that this requirement be changed to permit any type to serve as the basis of a new type.

4. No Free Union of Subset Types (E7).

a. <u>Ambiguity</u>, <u>Inconsistency</u>. There is a conflict with D5 (see comments there).

b. <u>Ambiguity</u>. The meaning of free union is ambiguous. It could be either a variation on discriminated union or another name for equivalence.

Section VII. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed (F1).

Ambiguity. It is not clear to what degree this requirement must be supported. Several possibilities exist. One possibility is to provide block structure with automatic variables. In languages having this feature, allocated variables may not be accessible in inner scopes if the same name is used for a local variable in that inner scope. Another possibility is to provide own (static) variables. Allocations of these variables persist longer than their scopes of access. A third possibility is to provide full separation. In this case, the scope of allocation is specified completely independent of the scope of access. The first case is supported by virtually all languages which are derivitives of ALGOL 60. The second case introduces a facility which is known to be error-prone and which is not needed in a language with user-defined types. The third case is a facility which is useful, but no known languages support such a facility without introducing considerable complexity.

2. Limiting Access Scope (F2).

Ambiguity, Overspecification, Comment. "Limited access specified in a type definition is necessary to guarantee that changes to data representations ... which do not affect calling programs are in fact safe." This intention seems to be that representations can be hidden in user-defined types. Limiting namescopes is not the only possible way of satisfying this intention. Also note that this condition is not sufficient to guarantee the ability to arbitrarily change a representation.

Section VIII. CONTROL STRUCTURES

1. Kinds of Control Structures (G1).

a. Disagreement. "...selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs." We question the advisability of this approach on four grounds:

(1) Without syntax extension (precluded by H2), a user is constrained by the existing notations (e.g., procedure invocation form) to contrive control structures of a funny-looking kind.

(2) It is unclear whether a user-defined control structure offers enough advantage to outweigh the attendant loss of program readability.

(3) Whether a comprehensive range of control structures can be achieved "by extension" at a tolerable level of inefficiency is probably still a research topic.

(4) The extension machinery itself will add, perhaps sizably, to the cost of compilers.

It seems preferable, at this time, to specify a reasonable set of control structures as primitives, and make no attempts to go further until a clear need can be demonstrated.

b. Ambiguity. Does the requirement for pseudo-parallel processing imply a coroutine-like mechanism?

2. The Go To (G2).

Ambiguity. "The 'GO TO' will be limited to... labels at the same scope level." Exactly what constitutes a scope level is subject to various interpretations: (a) a begin block (definitely!), (b) an alternative of an if statement (absolutely no), (c) a case statement (maybe), and (d) a repeat statement (maybe).

3. Conditional Control (G3).

Disagreement. The requirement that each IF statement require both a THEN and an ELSE clause is not justified. For readability, it is frequently useful to omit the ELSE clause and have control transfer to the next statement.

4. Iterative Control (G4).

a. Ambiguity, Overspecification. "The iterative control structure should permit the termination condition to appear anywhere in the loop...." Although this generalization of the WHILE and UNTIL statements appears useful, it must nevertheless be recognized that it can work in the context of nested loops only when the test for termination specifies explicitly the containing loop structure to which it belongs. It is unclear how the IF-statement with EXIT fails to give adequate capability in this respect.

b. Inconsistency. "...a specialized control structure should be provided for that purpose (e.g., FORTRAN DO or Algol-60 for)." This statement is in conflict with the sentence in G1 which states: "By these criteria, the Algol-60 for would be undesirable...".

c. Inconsistency, Overspecification. Whether or not a loop control variable should be local to the loop body, and whether its value should be specified and available following loop termination seem largely matters of taste and efficiency. If, as the Tinman states, "Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aid program clarity or correctness...", then it is puzzling why in the following sentence it is asserted that "at loop termination it should be possible to pass their value...out of the loop".

d. Comment. It seems desirable to make an additional requirement that the value of a loop control variable may not be modified by any statements in the body of the loop.

5. Routines (G5).

Disagreement. "It will not be possible to define procedures written within the body of a recursive procedure." "A major...cost...is the need for...'display'

registers...." This restriction seems unnecessarily strong. Displays are not the only way of removing this restriction. Other techniques which limit run-time overhead to exactly those cases where this restriction is violated are possible. We recommend that this restriction be dropped.

6. Parallel Processing (G6).

Comment. The terminology used in this section does not agree with standard practice. For example, "simultaneous activations of a given parallel process".

7. Exception Handling (G7).

a. Comment. The motivation for the restriction of transfers of control "forward in the program" was not clear.

b. Overspecification. There are many reasonable alternative exception-handling mechanisms. The requirements here should be less specific.

8. Synchronization and Real Time (G8).

Overspecification. Asynchronous hardware interrupts are different from synchronous exception conditions. They need not necessarily be handled by the same mechanism.

Section IX. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics (H1).

<u>Inconsistency</u>. "The source language... will be based on conventional forms." This may be in conflict with H10: "No language defined symbols... will have essentially different meanings". As an example, consider "*". This is conventionally used for both multiplication and to designate unresolved array dimensions.

2. No Syntax Extensions (H2).

a. <u>Inconsistency</u>. "The user will not be able to... define new infix operator precedences." This is in conflict with E2: "Defined types will be indistinguishable from built-in types."

b. <u>Ambiguity, Disagreement</u>. See our comments under C2.

c. <u>Ambiguity</u>. If new operators can be defined (E1) but not new precedences (H2), what are the parsing rules for these new operators?

3. Identifiers and Literals (H4).

<u>Comment</u>. "The language should require separate quoting of each line of a long literal." It is recommended that the appropriate connective operator (concatenation, for strings) be required between components of a multi-part literal to explicitly describe intent. This also avoids special rules about how two consecutive tokens are to be treated in certain special cases.

4. Key Words (H6).

<u>Ambiguity, Inconsistency</u>. "Key words will be reserved... and will not be usable... where an identifier can be used." Consider the key word TRUE. This can be used in places where a Boolean identifier can be used.

5. Comment Conventions (H7).

<u>Comment</u>. The following observation is admittedly sardonic, but it is somewhat amusing that the Tinman's suggested comment convention is claimed to "nearly" meet the

Tinman's requirements: if satisfying the comment requirements is difficult, what hope is there that the difficult issues can be satisfactorily addressed?

6. Unmatched Parentheses (H8).

Comment. "Failure to require proper parentheses matching makes it more difficult to write correct programs." This is incorrectly polarized: what this makes difficult is the detection of incorrect programs. We agree with the requirement, in any case.

7. Uniform Referent Notation (H9).

Disagreement. "There will be a uniform referent notation." Although we agree with the intent of this requirement, we believe that it may be exceedingly difficult to meet completely. For example, consider an array A and the assignment

A(I) := A(I) + 1

Since uniform referent applies, it should be possible to change A from an array to a function. Note that it now must be possible to have left-hand-side functions (like PL/I pseudo variables) which are user-definable. This kind of function is quite difficult to realize and results in a much more complex language. We recommend that this requirement be softened.

8. Consistency of Meaning (H10).

a. Comment. "No language defined symbols appearing in the same context should have essentially different meanings." Another wording problem: what is obviously intended is that a language symbol should not have essentially different meanings in different contexts.

b. Inconsistency. See our comments under H1.

Section X. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic (I1).

<u>Ambiguity</u>, <u>Comment</u>. There seems to be an unfortunate entanglement of ideas here. Apparently what is intended in I1 is the statement that implementation-dependency should be avoided wherever possible. We fully concur with this objective. However, this is a totally distinct issue from <u>defaults</u>.

2. Compile-Time Variables (I3).

<u>Ambiguity</u>, <u>Comment</u>. If the requirement sentences of I3 were replaced by the phrase occurring later ("the programmer should be able to specify the intended object machine configuration"), the meaning of the section would be clearer. Also, referring to these specifications as "compile-time variables" is likely to cause confusion; such specifications are in fact compile-time <u>constants</u>.

3. Translator Restrictions (I6).

a. <u>Comment</u>. The requirement that restrictions such as the number of array dimensions be specified in the language definition is in conflict with D5 ("There should not be any arbitrary restrictions on the structure of data").

b. <u>Disagreement</u>. Specifying independent bounds on such translator parameters as the length of identifiers and the number of identifiers is overly restrictive; for example, it may be the total number of characters required by the identifiers which is the critical factor, not the maximum length or the number of identifiers.

c. <u>Disagreement</u>. In view of the wide variety of hardware configurations and memory capacities available within the DoD, we do not feel that reasonable bounds of the kind intended by I6 are feasible.

Section XI. EFFICIENT OBJECT REPRESENTATION AND MACHINE DEPENDENCIES

1. Efficient Object Code (J1).

<u>Comment</u>. The sentence "The language should not force programs to require greater generality than they need" is hard to understand.

2. Optimizations Do Not Change Program Effect (J2).

<u>Comment</u>. "Any optimization performed by the translator will not change the effect of the program." The concept "change the effect" must be given a <u>very</u> careful definition. While the rules of algebra show that x + (y - z) is identical to (x + y) - z, ordinary floating-point hardware produces different results in some cases. For example, suppose x has a value of Ø.1 and y and z have identical values of 1Ø**2Ø. In the first case, the result is exactly x, while in the second case, the value is exactly zero. We would argue that preservation of the <u>algebraic</u> effect is sufficient under "optimization", even if the numerical result is not exactly preserved. Otherwise, too many useful optimizations would be disallowed. Another such "change the effect" question is illustrated by x := Ø; ...; y := z/x; with a mundane compiler, the division by zero will not be detected before run time. At run time, the appropriate exception will be signalled, corresponding recovery action (presumably) performed, and execution would continue. If a more clever compiler were to notice (at compile time) the inevitability of the division by zero, and issue an error indication and prevent execution, would this constitute a "change of effect"?

3. Machine Language Insertions (J3).

<u>Overspecification</u>. "The machine language insertions should be permitted only within the body of compile-time conditional statements...." The goal is correct: encapsulation of machine-dependency. However, the stated requirement seems limiting, if it precludes other suitable encapsulated forms. Procedure-level encapsulation appears particularly attractive as an alternative.

4. Object Representation Specifications (J4).

a. Ambiguity. "These descriptions [of the representation of composite data structures]... will be distinct from the logical description." The meaning of "logical description" is not clear. The intent of this requirement seems to be that the representation of a data type should be hidden in a way so that changes to the representation do not affect those parts of the program which use (rather than define) the data type.

b. Ambiguity, Disagreement. If the representation is not specified, "the object representation will be optimal as determined by the translator". First, at least some indication of the requirements for the representation must be specified (e.g., the value set of the type being defined). Second, the translator should not be required to make an optimal choice since this is exceedingly difficult.

5. Open and Closed Routine Calls (J5).

a. Comment. "An open and a closed routine of the same description will have identical semantics." This must not be construed to mean that any closed routine can be changed to an open routine. The difficulty can be seen by considering an open recursive routine.

b. Comment. "Open routine capability is especially important for machine language insertions." This is sometimes true, of course. Nonetheless, the historical preponderance of machine language coupled with high-level language has been in the area of function libraries, where speed and space considerations led to closed machine language routines.

c. Comment. "Thus, an open routine should differ from a syntax macro...." An open routine and a closed routine of the same description have identical semantics. Thus, open routines differ from syntax macros in precisely the way that closed routines do.

Section XII. PROGRAM ENVIRONMENT

1. Operating System Not Required (K1).

Ambiguity, Inconsistency. What is meant by an operating system? Are any run-time support routines considered to be in an operating system? If so, then this requirement conflicts with G6, which requires a parallel processing capability and with B10, which requires I/O operations.

2. Program Assembly (K2).

a. Comment. The distinction drawn between separate definition and separate compilation is a good point. General practice seems to allow us to be careless with our terminology in this area.

b. Disagreement, Overspecification. It is by no means agreed that "...separate compilation increases the difficulty and expense of the interface validations...". Anyway, this is one of many implementation (not language) considerations which appear in the Tinman, but perhaps should not.

3. Assertions and Other Optional Specifications (K5).

Inconsistency. "[Assertions] will have the status of comments." This is in conflict with H7, which requires only a single comment convention.

Section XIII. TRANSLATORS

1. No Subset Implementations (L2).

Comment. "Every translator for the language should implement the entire language." Although there can be little argument that this is a laudable goal, these Tinman requirements themselves threaten to make the goal impractical. (For example, see our comments on C1 and C4.) Paraphrasing Prof. Hoare: a prudent procedure to follow would be to strive for a language in which subsetting is not necessary, and to plan for subsets.

2. Low Cost Translation (L3).

a. Ambiguity, Disagreement. "The translator should minimize compile-time costs." Not only is this sentence disputed in two places in the paragraphs which follow it, but it does not really convey any information unless the set of constraints which must be satisfied is specified. Obviously, more time will be required during compilation to produce better object code, which sometimes will be strongly desirable.

b. Comment. "...trivial programs [should] compile and run in trivial time." While this goal seems (superficially) to be correct, it is nonetheless certain that the amount of DoD software budget spent on "trivial" programs is insignificant. If it should prove convenient, for some reason, to compile "trivial" programs ineffectively in order to make some gain in an area of high software spending, that alternative seems vastly superior. We would all like to do well in every area. But we must not be blind to the necessity of making trades to get the maximum payoff in places where it really matters. Example: programs need to be easy to read and comprehend with higher importance than they need to be easy to write.

3. Many Object Machines (L4).

Comment. "...makes the full power of an existing translator available at the cost of producing an additional code generator." Not to be overlooked are the related needs to supply other target-machine-dependent software, such as operating system interface, machine-dependent extensions and libraries, debugging aids, etc.

Section XIV. LANGUAGE DEFINITION, STANDARDS AND CONTROL

1. Existing Language Features Only (M1).

Ambiguity, Comment. "The language will be composed of features which are within the state of the art...." There is considerable disagreement in many areas as to what features are within the state of the art. Some might argue that certain of the Tinman required features are not within the state of the art (e.g., user-defined data types, uniform referents, extensive compile-time facilities). Note that even when several features are within the state of the art, a language in which they are combined may not be. In particular, research may be required on how to handle feature interactions.

2. Language Documentation Required (M3).

Overspecification. "The language will be defined as if it were the machine level language of an abstract digital computer." Although this kind of definition is certainly possible, there are other reasonable definitional approaches (e.g., axiomatic definition) that are ruled out by the requirement.